#### স্ট্রিং পরিচিতি

পাইথন এ অনেকগুলো ক্যারেক্টার নিয়ে স্ট্রিং গঠিত হয়, এখানে ক্যারেক্টার বলতে আলাদা কোনো ডাটা টাইপ নাই. ইউনিকোড ক্যারেক্টার এর সিকোয়েন্সে কে স্ট্রিং বলে. যাইহোক, পাইথন এ কোনো ক্যারেক্টার ডাটা টাইপ নাই. একটি সিঙ্গাল ক্যারেক্টার কে একটি স্ট্রিং হিসাবে গণ্য করা হয়, যার length হলো ১. স্কয়ারে ব্রাকেট ব্যবহার করে আমরা স্ট্রিং এর এলিমেন্টস এক্সেস করতে পারি.

**স্ট্রিং এর মেথড ও ফাঙ্শন**

পাইথন এ বিল্টিন মেথড এর একটি সেট রয়েছে যা আপনি স্ট্রিংগুলোতে ব্যবহার করতে পারেন.  নিচে মেথডগুলো সম্পর্কে সংক্ষেপে আলোচনা করা হলো:

**নোট:**সকল স্ট্রিং মেথডগুলি নতুন ভ্যালু রিটার্ন করে, তারা অরিজিনাল স্ট্রিং পরিবর্তন করে না.

|  |  |
| --- | --- |
| মেথড | বর্ণনা |
| capitalize() | প্রথম অক্ষরে কে বড় হাতের অক্ষরে রুপান্তর করে |
| center() | একটি কেন্দ্রীভূত (centerd) স্ট্রিং প্রদান করে |
| count() | একটি স্ট্রিং যে কোনো একটি ভ্যালু কতবার আছে সেই সংখ্যা রিটার্ন করে |
| encode() | একটি এনকোডেড স্ট্রিং রিটার্ন করে |
| find() | একটি সুনির্দিষ্ট মানের জন্য স্ট্রিং অনুসন্ধান করে এবং যেখানে এটি পাওয়া গেছে তার অবস্থান বলে দেয় |
| format() | স্ট্রিং ফরমেটিং এর কাজ করে |
| index() | একটি সুনির্দিষ্ট মানের জন্য স্ট্রিং অনুসন্ধান করে এবং যেখানে এটি পাওয়া গেছে তার অবস্থান (index number) বলে দেয় |
| lower() | একটি স্ট্রিং কে ছোট হাতের অক্ষরে রূপান্তর করে |
| title() | প্রত্যেকটা শব্দের প্রথম অক্ষরকে বড় হাতের অক্ষরে রূপান্তর করে |
| upper() | একটি স্ট্রিংকে বড় হাতের অক্ষরে রূপান্তর করে |

**একই সাথে একাধিক ভ্যারিয়েবল**

* মাল্টিপল এসাইনমেন্ট
* পাইথনে আমরা চাইলে এক সাথে একের অধিক ভ্যারিয়েবলে একই ডেটা এসাইন করতে পারি। যেমনঃ
* mahfuz, nasim, minhaz = 3.50  
  আমরা চাইলে একই সাথে একাধিক ভ্যারিয়েবলে একাধিক মান এক সাথে এসাইন করতে পারি। যেমনঃযেখানে ovi, niloy, asif নামক তিনতে ভ্যারিয়েবলে প্রতিটাতে 3.99 ভ্যালুটি এসাইন হবে।
* pi, name, number = 3.1416, 'mahfuz', 999
* যেখানে আমরা তিনটে ভ্যারিয়েবলে এক সাথে তিনতে আলাদা আলাদা ভ্যালু এসাইন করেছি। যেখানে pi = 3.1416, name = “Guido van Rossum” এবং  number = 999।

#### লিস্ট পরিচিতি

লিস্ট শব্দের বাংলা অর্থ তালিকা। আমাদের বোধহয় ব্যখ্যা করার দরকার পড়ে না তালিকা কি জিনিস । পাইথনেও লিস্ট একই কাজ করে । সহজ কথায় লিস্ট হল কতগুলো আইটেমের একটি তালিকা । অনেক প্রোগ্রামিং ল্যাঙ্গুয়েজে লিস্ট ডিক্লেয়ার করার সময় বলে দিতে হয় লিস্টের আইটেমগুলোর টাইপ কি হবে, পাইথনে তার দরকার পড়ে না । একটি লিস্টের আইটেমগুলো বিভিন্ন টাইপের হতে পারে ।

কিভাবে লিস্ট ডিক্লেয়ার করব? (থার্ড) ব্রাকেটের ভিতরে কমা দিয়ে একেকটি আইটেম সেপারেট করে দিলেই লিস্ট তৈরি হয়ে যাবে ।

Ex: list = [1,2,3,4,5]

#### লিস্ট স্লাইসিং

স্লাইসিং এর মাধ্যমে একটি লিস্ট কে  ইচ্ছামতো ছোটো বড়ো করা যায় । এক্ষেত্রে ইনডেক্স এর ব্যাবহারটা বেশ সুবিধাজনক । লিস্ট স্লাইসিং একটি কমন প্রাকটিস এবং প্রোগ্রামারদের জন্য সমস্যা সমাধানের একটি কার্যকরী পদ্ধতি । বড় একটি লিস্ট থেকে যদি তুমি একটি ছোটো লিস্ট চাও, তাহলে এ পদ্ধতি টি খুবই কার্যকর  । এ ক্ষেত্রে সিম্পল স্লাইসিং অপারেটর ':' (কোলন) ব্যবহার হয় । এই অপারেটর এর মাধ্যমে ইউসার নির্ধারণ করতে পারে যে লিস্ট টি কোথা থেকে শুরু হবে এবং কোথায় শেষ হবে । এটি এক্সিস্টিং লিস্ট থেকে একটি নতুন লিস্ট রিটার্ন করে ।

#### লিস্ট এর মেথড ও ফাঙ্শন

পাইথন এ বিল্ট ইন মেথড এর একটি সেট রয়েছে, যেগুলোকে আমরা লিস্ট এর উপর বিভিন্ন অপারেশন এর  জন্য ব্যবহার করতে পারি, নিচে এগুলো সম্পর্কে সংক্ষেপে আলোচনা করা হলো:

append(), এটি লিস্ট এর শেষে একটি উপাদান যোগ করে.

remove(), এটি লিস্ট থেকে সব উপাদানগুলোকে দূর করে দেয়.

copy(), লিস্ট এর একটি কপি রিটার্ন করে.

count(). কোনো একটি নিদ্দিষ্ট উপাদান লিস্ট এ কতবার আছে সেই সংখ্যাটি প্রকাশ করে.

extend(), কোনো একটি লিস্ট এর উপাদান গুলোকে কারেন্ট লিস্ট এর শেষে যোগ করে.

index(), লিস্ট এর একটি নিদ্দিষ্ট উপাদান এর ইনডেক্স নাম্বার রিটার্ন করে.

insert(), লিস্ট এর একটি নিদ্দিষ্ট পজিশন এ একটি উপাদান কে রেখে দেয়.

pop(), লিস্ট এর একটি নিদ্দিষ্ট পজিশন থেকে একটি নিদ্দিষ্ট উপাদান কে দূর করে দেয়.

remove(), একটি নিদ্দিষ্ট উপাদান কে লিস্ট থেকে দূর করে দেয়.

reverse(), লিস্ট কে বিপরীতভাবে সাজিয়ে দেয়.

sort(), লিস্ট কে সাজিয়ে দেয়

#### টুপল পরিচিতি

লিস্টের মতই আরেকটি ডেটা স্ট্র্যাকচার হচ্ছে Tuple. কেউ উচ্চারন করে টাপল কেউ টুপল।

লিস্ট ডিক্লেয়ার করি আমরা স্কোয়ার ব্র্যাকেট দিয়ে।

l = [1,2,3,4,5]

t = (1,2,3,4,5)

টাপল ডিক্লেয়ার করি প্রথম ব্র্যাকেট দিয়েঃ

টাপলের ভ্যালু এক্সেস করাঃ

প্রথম ভ্যলু পেতেঃ

t = (1,2,3,4,5)

print(t[0])

দ্বিতীয় ভ্যালু পেতেঃ

t = (1,2,3,4,5)

print(t[1])

উপরের টাপলে শেষ ভ্যালু পেতে পারিঃ

t = (1,2,3,4,5)

print(t[4])

এভাবেও পেতে পারিঃ

t = (1,2,3,4,5)

print(t[-1])

যেমন t[0] এর ভ্যালু হচ্ছে 1, আমরা পরিবর্তন করে 10 করতে চাই, তাহলেঃএ কাজ গুলো লিস্টেও করা যায়। লিস্টের সাথে টাপলের প্রধান পার্থক্য হচ্ছে টাপল ইমিউটেবল। লিস্ট মিউটেবল। মিউটেবল মানে পরিবর্তন করা যায়। আর ইমিউটেবল মানে পরিবর্তন করা যায় না।

t = (1,2,3,4,5)

t [0] = 10

লিস্টে সহজেই আমরা তা করতে পারিঃকিন্তু তা কাজ করবে না। ইরর দেখাবে।

l = [1,2,3,4,5]

l [0] = 10

print(l)

টাপল ইটারেশনঃ

t = (1,2,3,4,5)

for item in t :

      print(item)

টাপল থেকে লিস্ট অনেক শক্তিশালি এবং অনেক গুলো সুবিধে। তারপর ও আমরা টাপল ব্যবহার করব যেখানে আমাদের কনটেইনারের ডেটা পরিবর্তন করতে হবে না।

#### সেট পরিচিতি

# **পাইথনের ডেটা স্ট্রাকচার – সেট**

সেট আমরা অনেকেই পড়েছি, না পড়লে নবম-দশম শ্রেণীর গণিত বই থেকে পড়ে নিতে হবে। তবে এখনই সেটি করা দরকার নেই, সেটের মূল ধারণাগুলো আমি আলোচনা করবো।

সেট হচ্ছে বিভিন্ন জিনিসের সমাবেশ, ইংরেজিতে বলা যায় collection of items। যেমন, আমি যদি বলি যে আমার টেবিলে এখন আছে একটি ল্যাপটপ, একটি মোবাইল, একটি গ্লাস, একটি নোটবুক, একটি কলম, তাহলে এগুলোকে একটি সেট বলা যায়। সেটের ভেতরের জিনিসগুলো দ্বিতীয় বন্ধনীর ভেতরে লিখতে হয়। দ্বিতীয় বন্ধনীকে ইংরেজিতে বলে curly braces।

{laptop, cellphone, glass, notebook, pen}

আবার আমরা যেমন বলি, সপ্তম শ্রেণীর পাঠ্যবইয়ের একটি সেট। সেটের উপাদানগুলো যেকোনো ক্রমে লিখা যায়। যেমন, সপ্তম শ্রেণীর পাঠ্যবইয়ের সেটে আমরা বাংলা, ইংরেজি, গণিত, সমাজবিজ্ঞান … এভাবে লিখতে পারি, কিংবা, ইংরেজি, বাংলা, সমাজবিজ্ঞান, গণিত … এভাবেও লেখা যায়, তাতে কোনো সমস্যা নেই, সেট একই থাকছে। তবে সেটের কোনো উপাদান একাধিকবার লেখা যায় না। আমার টেবিলে যদি দুটি কিংবা আরো বেশি কলম থাকতো, তাহলেও আমি সেটের মধ্যে pen একবারই লিখতাম।

দুটো সেটের সবগুলো উপাদান মিলে নতুন সেট তৈরি করা যায়। এটিকে বলে ইউনিয়ন (union) করা, মানে, ইউনিয়ন এখানে একটি অপারেশন। যেমন : A = {1, 2, 3, 4, 5} ও B = {2, 4, 6, 8, 10} দুটি সেট হলে A ইউনিয়ন B হবে : {1, 2, 3, 4, 5, 6, 8, 10}। আবার দুটো সেটের সাধারন উপাদানগুলো অর্থাৎ যেসব উপাদান দুটো সেটেই আছে, তাদেরকে নিয়ে একটি নতুন সেট তৈরি করা যায়। যে অপারেশনের মাধ্যমে এটি করা হয়, তাকে বলে ইন্টারসেকশন (intersection)। A ও B-এর ইন্টারসেকশন করলে আমরা পাবো : {2, 4}।

পাইথনে সেট নিয়ে কাজ করার জন্য সেট নামেই একটি ডেটা স্ট্রাকচার আছে। যেমন আমরা একটি ফাঁকা সেট তৈরি করতে পারি এভাবে : A = set()। নিচে উদাহরণ দেওয়া হলো :

>>> A = set()

>>> A

set()

>>> type(A)

<class 'set'>

আবার আমাদের কাছে যদি ইতিমধ্যেই কিছু উপাদান থাকে, সেগুলো থেকেও সেট তৈরি করা যায়। যেমন :

>>> items = {"pen", "laptop", "cellphone"}

>>> items

{'cellphone', 'pen', 'laptop'}

>>> type(items)

<class 'set'>

>>>

আবার আমাদের কাছে যদি কোনো লিস্ট বা টাপল থাকে, সেই লিস্ট বা টাপল থেকেও সেট তৈরি করা যায়।

>>> li = [1, 2, 3, 4]

>>> tpl = (1, 2, 3)

>>> A = set(li)

>>> A

{1, 2, 3, 4}

>>> B = set(tpl)

>>> B

{1, 2, 3}

>>> type(A)

<class 'set'>

>>> type(B)

<class 'set'>

>>>

স্ট্রিং থেকেও সেট তৈরি করা যায়, সে ক্ষেত্রে স্ট্রিংয়ের প্রতিটি অক্ষর সেটের একটি উপাদান হবে। আর সেটের ক্ষেত্রে কিন্তু একটি জিনিস মনে রাখতে হবে, সেটের উপাদান আগে-পরে থাকতে পারে, কোনো ক্রম (order) অনুসরণ করতে হয় না।

>>> A = set("Bangladesh")

>>> A

{'e', 'g', 'h', 'a', 's', 'l', 'B', 'd', 'n'}

>>> type(A)

<class 'set'>

>>> B = set("Sri Lanka")

>>> B

{'i', 'k', 'L', ' ', 'r', 'a', 'S', 'n'}

>>> type(B)

<class 'set'>

>>>

কোনো জিনিস একটি সেটের সদস্য কি না, সেটি আমরা বের করতে পারি এভাবে :

>>> A = {1, 2, 3}

>>> 1 in A

True

>>> 2 in A

True

>>> 5 in A

False

এখন, আমার যদি দুটি সেট থাকে, আমি সেই দুটি সেটের ওপর বিভিন্ন অপারেশন চালাতে পারি। আমরা যদি চাই যে, A ও B দুটি সেটের সাধারণ উপাদানগুলো নিয়ে একটি সেট C তৈরি করবো, তাহলে আমরা লিখতে পারি : C = A & B (মানে ইন্টারসেকশন করা)। আর আমরা যদি ইউনিয়ন করতে চাই, অর্থাৎ, A কিংবা B যেকোনো একটি সেটের সদস্য (সেসব সদস্য আবার দুটো সেটের মধ্যেও থাকতে পারে), তাহলে লিখবো C = A | B। আবার আমরা যদি চাই, A কিংবা B, যেকোনো একটি সেটে আছে কিন্তু একই সঙ্গে দুটি সেটে নেই, সেসব সদস্য পেতে, তখন লিখবো C = A ^ B। আর যদি এমন হয় যে, আমরা চাই যেসব সদস্য A সেটে আছে কিন্তু B সেটে নেই, তাদের পাওয়া যাবে এভাবে : C = A – B। নিচের উদাহরণ দেখলে আমরা বুঝতে পারবো :

>>> A = {1, 2, 3, 4, 5}

>>> B = {2, 4, 6, 8}

>>> C = A & B

>>> C

{2, 4}

>>> C = A | B

>>> C

{1, 2, 3, 4, 5, 6, 8}

>>> C = A ^ B

>>> C

{1, 3, 5, 6, 8}

>>> C = A - B

>>> C

{1, 3, 5}

>>> C = B - A

>>> C

{8, 6}

#### ডিক্শনারি পরিচিতি

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

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

1

my\_marks = {"Bengali": 80, "English": 85, "Math": 90}

আবার ফাকা ডিকশনারি তৈরির জন্য এভাবে লিখলেই সেটি ইনিসিয়ালাইজ হয়ে যায় - my\_dictionary = {}

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

উদাহরণ,

1

my\_marks = {"Bengali": 80, "English": 85, "Math": 90}

2

print(my\_marks["Math"])

আউটপুট,

1

90

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

1

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

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

1

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

2

print(my\_marks["Math"])

যার আউটপুট আসবে,

1

[60, 74, 58]

কিন্তু নিচের মত একটি ডিকশনারি হতে পারে না -

1

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

এর আউটপুট হবে,

1

Traceback (most recent call last):

2

File "<stdin>", line 1, in <module>

3

TypeError: unhashable type: 'list'

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

1

my\_marks = {True : "Bengali"}

**ডিক্শনারিতে আইটেম যোগ বা ডিলেট**

**আপডেট ডিকশনারি অথবা ডিলিট ডিকশনারি**

**ডিকশনারি তে নতুন আইটেম যুক্ত করতেঃ**

|  |  |
| --- | --- |
| 1  2  3 | friends = {'Shuvo':15, 'Tuhin':13, 'Tanvir':14, 'Dipu' :13 , 'Siyam':14 }  friends ['Mahir'] = 16  print (friends) |

**কোন আইটেম আপডেট করতেঃ**

যদি আপনার ফ্রেন্ডদের বয়স বেড়ে যায়, তাহলে তাদের বয়স তো আপডেট করতে হবে, তাই না?

|  |  |
| --- | --- |
| 1  2  3 | friends = {'Shuvo':15, 'Tuhin':13, 'Tanvir':14, 'Dipu' :13 , 'Siyam':14 }  friends ['Tuhin'] = 14  print (friends) |

এখানে Tuhin এর বয়স আপডেট করেছি।

**ডিকশনারি থেকে কোন আইটেম ডিলেট করতেঃ**

|  |  |
| --- | --- |
| 1  2  3  4 | friends = {'Shuvo':15, 'Tuhin':13, 'Tanvir':14, 'Dipu' :13 , 'Siyam':14 }  print(friends)  del friends[' Shuvo']  print(friends) |

**সব গুলো আইটেম ডিলেট করতেঃ**

|  |  |
| --- | --- |
| 1  2  3  4 | friends = {'Shuvo':15, 'Tuhin':13, 'Tanvir':14, 'Dipu' :13 , 'Siyam':14 }  print(friends)  friends.clear();  print(friends) |

এখন প্রিন্ট করে দেখব কোন আইটেম নেই।

**আমরা চাইলে সম্পুর্ণ ডিকশনারিই ডিলেট করে দিতে পারি এভাবেঃ**

|  |  |
| --- | --- |
| 1  2 | friends = {'Shuvo':15, 'Tuhin':13, 'Tanvir':14, 'Dipu':13, 'Siyam':14}  del friends |

**len()**  
len() দিয়ে একটা ডিকশোনারির লেন্থ বের করা যায়। যেমনঃ

|  |  |
| --- | --- |
| 1  2 | friends = { 'Shuvo ':15, 'Tuhin':13, 'Tanvir':14, 'Dipu' :13 , 'Siyam':14 }  print(len(friends)) |

যা প্রিন্ট করবে 5।

#### ডিক্শনারি কিভাবে আপডেট বা মার্জ করবো ?

কতগুলো নিয়ম ব্যবহার করে আমরা পাইথন এ ডিকশনারি আপডেট বা মার্জ করতে পারি:

মেথড ১: ইটারেশন এর মাধ্যমে.

d3 = d1.copy()for key, value in d2.items():

d3[key] = value

print(d3)

**Output:**

{'India': 'Delhi',

'Canada': 'Ottawa',

'United States': 'Washington D. C.',

'France': 'Paris',

'Malaysia': 'Kuala Lumpur'}

মেথড ২: আপডেট() মেথড ব্যবহার করে.

d4 = d1.copy()d4.update(d2)

print(d4)

**Output:**

{'India': 'Delhi',

'Canada': 'Ottawa',

'United States': 'Washington D. C.',

'France': 'Paris',

'Malaysia': 'Kuala Lumpur'}

মেথড ৩: আনপ্যাকিং অপারেটর (\*\*) ব্যবহার করে.

d5 = {\*\*d1, \*\*d2}

print(d5)

**Output:**

{'India': 'Delhi',

'Canada': 'Ottawa',

'United States': 'Washington D. C.',

'France': 'Paris',

'Malaysia': 'Kuala Lumpur'}

মেথড ৪: collection.Chainmap() ব্যবহার করে.

from collections import ChainMap

d6 = ChainMap(d1, d2)

print(d6)

**Output:**

ChainMap({'Canada': 'Ottawa',

'India': 'Delhi',

'United States': 'Washington D. C.'},

{'France': 'Paris',

'Malaysia': 'Kuala Lumpur'})

মেথড ৫: 2nd ডিকশনারি কে আনপ্যাক করে

d7 = dict(d1, \*\*d2)

print(d7)

**Output:**

{'India': 'Delhi',

'Canada': 'Ottawa',

'United States': 'Washington D. C.',

'France': 'Paris',

'Malaysia': 'Kuala Lumpur'}

মেথড ৬: মার্জ অপারেটর ( | )ব্যবহার করে.

d8 = d1 | d2

print(d8)**Output:**

{'India': 'Delhi',

'Canada': 'Ottawa',

'United States': 'Washington D. C.',

'France': 'Paris',

'Malaysia': 'Kuala Lumpur'}

#### এরিথমেটিক, এসাইনমেন্ট, কম্পারিজন অপারেটরস

# **পাইথন অপারেটর**

এই অধ্যায়ে আপনি পাইথনে ব্যবহৃত বিভিন্ন অপারেটর, এগুলোর সিনট্যাক্স এবং এগুলোকে কিভাবে আপনার নিজের প্রোগ্রামে প্রয়োগ করবেন তা জানবেন।

পাইথনে অপারেটর হলো বিশেষ ধরনের প্রতীক যা গাণিতিক এবং যৌক্তিক (logical) হিসাব-নিকাশে ব্যবহৃত হয়। অপারেটর(**+, -, \*** ইত্যাদি) যে ভ্যালুকে অপারেট করে তাকে অপারেন্ড বলা হয়।

**উদাহরণস্বরূপঃ**

>>> 3+7

#Output: 10

এখানে + একটি অপারেটর যা যোগের কাজে ব্যবহৃত হয়েছে। 3 ও 7 অপারেন্ড এবং 10 অপারেশনের আউটপুট।

**পাইথনে অপারেটরসমূহঃ**

## **এরিথমেটিক অপারেটর**

গাণিতিক যোগ-বিয়োগ, গুণ-ভাগ ইত্যাদিতে এরিথমেটিক অপারেটর(Arithmetic operator) ব্যবহৃত হয়।

|  |  |  |
| --- | --- | --- |
| পাইথনে এরিথমেটিক অপারেটরসমূহ | | |
| অপারেটর | অর্থ | উদাহরণ |
| + | দুটি অপারেন্ডের যোগ বা ইউনারি যোগ(unary plus) | x + y |
| - | বাম অপারেন্ড থেকে ডান অপারেন্ড বিয়োগ বা ইউনারী বিয়োগ | x - y |
| \* | দুটি অপারেন্ডের গুণ | x \* y |
| / | বাম অপারেন্ডকে ডান অপারেন্ড দিয়ে ভাগ(ফলাফল সব সময় দশমিক হবে) | x / y |
| % | মডিউলাস(Modulus) - বাম অপারেন্ডকে ডান অপারেন্ড দিয়ে ভাগ করে ভাগশেষ | x % y (x/y এর ভাগশেষ) |
| // | Floor division -ভাগশেষ বাদে পূর্ণ সংখ্যায় ভাগফল | x // y |
| \*\* | সূচক(Exponent)- ডান অপারেন্ড বাম অপারেন্ড এর সূচক | x\*\*y (x এর সূচক y) |

### ****উদাহরণঃ**** পাইথনে এরিথমেটিক অপারেটর

x = 10

y = 4

print('x + y =',x+y)

# Output: x + y = 14

print('x - y =',x-y)

# Output: x - y = 6

print('x \* y =',x\*y)

# Output: x \* y = 40

print('x / y =',x/y)

# Output: x / y = 2.50

print('x // y =',x//y)

# Output: x // y = 3

print('x \*\* y =',x\*\*y)

# Output: x \*\* y = 10000

**কম্প্যারিজন অপারেটর**

দুই বা ততোধিক ভ্যালুর মধ্যে তুলনা করতে কম্প্যারিজন অপারেটর ব্যবহৃত হয়। কোনো শর্তের(condition) উপরে ভিত্তিকরে এটি হয় True অথবা False রিটার্ন করে।

|  |  |  |
| --- | --- | --- |
| পাইথনে কম্প্যারিজন অপারেটরসমূহ | | |
| অপারেটর | অর্থ | উদাহরণ |
| > | Greater Than - বামপক্ষ ডানপক্ষের চেয়ে বড় হলে True হবে। | x > y |
| < | Less Than - বাম পক্ষ্য ডানপক্ষের চেয়ে ছোট হলে True হবে। | x < y |
| == | Equal to - বামপক্ষ এবং ডানপক্ষ সমান হলে True হবে। | x == y |
| != | Not equal to - বামপক্ষ এবং ডানপক্ষ সমান না হলে True হবে। | x != y |
| >= | Greater than or equal to - বামপক্ষ ডানপক্ষের চেয়ে বড় বা সমান হলে True হবে। | x >= y |
| <= | Less than or equal to - বামপক্ষ ডানপক্ষের চেয়ে ছোট বা সমান হলে True হবে। | x <= y |

### ****উদাহরণঃ**** পাইথনে কম্প্যারিজন অপারেটর

x = 10

y = 15

print('x > y is',x>y)

# Output: x > y is False

print('x < y is',x<y)

# Output: x < y is True

print('x == y is',x==y)

# Output: x == y is False

print('x != y is',x!=y)

# Output: x != y is True

print('x >= y is',x>=y)

# Output: x >= y is False

print('x <= y is',x<=y)

# Output: x <= y is True

## **এসাইনমেন্ট অপারেটর**

ভ্যারিয়েবলে ভ্যালু এসাইন বা আরোপ করার জন্য পাইথনে এসাইনমেন্ট অপারেটর(Assignment operator) ব্যব্যহৃত হয়।

a = 10 একটি সাধারণ এসাইনমেন্ট অপারেটর যা ডানপাশের ভ্যালু 10 কে বামপাশের ভ্যারিয়েবল a তে এসাইন করে।

পাইথনে বিভিন্ন ধরণের অপারেটর আছে যেমন- a += 5 ভ্যারিয়েবলের সাথে ভ্যালু যোগ করে যোগফল পরবর্তীতে একই ভ্যারিয়েবলে এসাইন করে। এটি a = a + 5 এর সমতুল্য।

|  |  |  |
| --- | --- | --- |
| পাইথনে এসাইনমেন্ট অপারেটর | | |
| অপারেটর | উদাহরণ | সমতুল্য |
| = | x = 5 | x = 5 |
| += | x += 5 | x = x + 5 |
| -= | x -= 5 | x = x - 5 |
| \*= | x \*= 5 | x = x \* 5 |
| /= | x /= 5 | x = x / 5 |
| %= | x %= 5 | x = x % 5 |
| //= | x //= 5 | x = x // 5 |
| \*\*= | x \*\*= 5 | x = x \*\* 5 |
| &= | x &= 5 | x = x & 5 |
| |= | x |= 5 | x = x | 5 |
| ^= | x ^= 5 | x = x ^ 5 |
| >>= | x >>= 5 | x = x >> 5 |
| <<= | x <<= 5 | x = x << 5 |

#### লজিকাল, আইডেন্টিটি ও মেম্বারশিপ অপারেটরস

## **লজিক্যাল অপারেটর**

পাইথনে and, or এবং not অপারেটরসমূহকে লজিক্যাল অপারেটর(Logical operator) বলা হয়।

|  |  |  |
| --- | --- | --- |
| পাইথনে লজিক্যাল অপারেটরসমূহ | | |
| অপারেটর | অর্থ | উদাহরণ |
| and | উভয় অপারেন্ড true হলে True | a and b |
| or | যেকোনো একটি অপারেন্ড true হলে True | a or b |
| not | অপারেন্ড false হলে True | not a |

### ****উদাহরণঃ**** পাইথনে লজিক্যাল অপারেটর

a = True

b = False

print('a and b is',a and ab)

# Output: a and b is False

print('a or b is',a or b)

# Output: a or b is True

print('not a is',not a)

# Output: not a is False

### ****আইডেন্টিটি অপারেটর****

পাইথনে is এবং is not হচ্ছে আইডেন্টিটি অপারেটর। দুটি ভ্যলু(অথবা ভ্যারিয়েবল) মেমোরির একই অংশে অবস্থিত কিনা চেক করার জন্য এগুলো ব্যবহৃত হয়।

দুটি ভ্যারিয়েবল সমান হওয়ার অর্থ এই নয় যে তারা আইডেন্টিকাল। সুতরাং দুটি ভ্যালু সমান হলে তারা আইডেন্টিকাল নাও হতে পারে।

|  |  |  |
| --- | --- | --- |
| পাইথনে আইডেন্টিটি অপারেটর | | |
| অপারেটর | অর্থ | উদাহরণ |
| is | অপারেনড আইডেন্টিকাল হলে True হবে(একই অবজেক্টকে রেফার করে)। | x is True |
| is not | অপারেনড আইডেন্টিকাল না হলে True (একই অবজেক্টকে রেফার করে না) হবে। | x is not True |

### ****উদাহরণঃ**** পাইথনে আইডেন্টিকাল অপারেটর

x1 = 5

y1 = 5

x2 = 'Hello'

y2 = 'Hello'

x3 = [1,2,3]

y3 = [1,2,3]

print(x1 is not y1)

# Output: False

print(x2 is y2)

# Output: True

print(x3 is y3)

# Output: False

এখানে আমরা দেখতে পাচ্ছি যে, x1 এবং y1 একই ভ্যালুর পূর্ণ সংখ্যা, সুতরাং এরা সমান এবং আইডেন্টিকাল। একইভাবে স্ট্রিং x2 এবং y2 সমান এবং আইডেন্টিকাল।

কিন্তু x3 এবং y3 হচ্ছে লিস্ট। এরা সমান কিন্তু আইডেন্টিকাল নয়। যেহেতু লিস্ট পরিবর্তনশীল তাই এরা সমান হওয়া সত্ত্বেও ইন্টারপ্রেটার এদেরকে মেমোরির বিভিন্ন জায়াগায় স্থান দেয়।

### 

### ****মেম্বারশিপ অপারেটর****

পাইথনে in এবং not in হচ্ছে মেম্বারশিপ অপারেটর(Membership operator)। কোনো ভ্যালু বা ভ্যারিয়েবল [string](https://www.sattacademy.com/python/python-operators.php), [list](https://www.sattacademy.com/python/python-operators.php), [tuple](https://www.sattacademy.com/python/python-operators.php), [set](https://www.sattacademy.com/python/python-operators.php) এবং [dictionary](https://www.sattacademy.com/python/python-operators.php) ক্রমের মধ্যে আছে কি না যাচাই করার জন্য এগুলো ব্যবহৃত হয়।

ডিকশনারিতে ভ্যালুর পরিবর্তে কোনো কী(key)-এর উপস্থিতি আছে কিনা যাচাই করা হয়।

|  |  |  |
| --- | --- | --- |
| পাইথনে মেম্বারশিপ অপারেটর | | |
| অপারেটর | অর্থ | উদাহরণ |
| in | ক্রম(sequence)-এর মধ্যে ভ্যলু/ভ্যারিয়েবল থাকলে True হবে। | 5 in x |
| not in | ক্রম(sequence)-এর মধ্যে ভ্যলু/ভ্যারিয়েবল না থাকলে True হবে। | 5 not in x |

### ****উদাহরণঃ**** পাইথনে মেম্বারশিপ অপারেটর

x = 'Hello world'

y = {1:'a',2:'b'}

print('H' in x)

# Output: True

print('hello' not in x)

# Output: True

print(1 in y)

# Output: True

print('a' in y)

# Output: False

এখানে x এর মধ্যে 'H' আছে কিন্তু x এর মধ্যে 'hello' নাই (মনে রাখবেন, পাইথন কেস-সেন্সিটিভ)। একইভাবে ডিকশনারিতে y-এ 1 হচ্ছে কী 'a' হচ্ছে ভ্যালু। , 'a' in y এর ফলাফল False হয়।

#### কন্ডিশনাল স্টেটমেন্ট

# **পাইথন if স্টেটমেন্ট**

সিদ্ধান্ত গ্রহণের জন্য প্রোগ্রামিং এ সচরাচর কন্ডিশনাল(conditional) স্টেটমেন্ট ব্যবহৃত হয়। এই অধ্যায়ে আপনি if কন্ডিশনাল স্টেটমেন্ট ব্যবহার করে সিদ্ধান্ত গ্রহণ করা শিখবেন।

## **কন্ডিশনাল স্টেটমেন্ট**

পাইথন প্রোগ্রামিং এ তিন ধরণের if কন্ডিশনাল(conditional) স্টেটমেন্ট রয়েছেঃ

> if

> if-else

> if-elif-else

## **পাইথনে if স্টেটমেন্ট এর সিনট্যাক্স**

if testExpression:

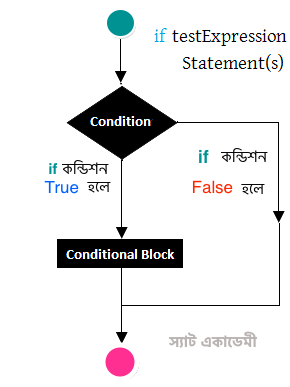
statement(s)

**সিনট্যাক্স এর ব্যাখ্যা**

* প্রোগ্রাম প্রথমে **testExpression** কে মূল্যায়ন করে এবং এটি যদি True হয় তাহলে statement(s) সম্পাদিত হয়।
* যদি textExpression এর ভ্যালু False হয় তাহলে statement(s) সম্পাদিত হয় না।
* পাইথনে if স্টেটমেন্ট এর বডি(Body)-কে ইন্ডেন্টেশন(indentation)-এর মাধ্যমে লিখতে হয়। সুতরাং Body শুরু হয় ইন্ডেন্টেশনের মাধ্যমে এবং প্রথম যেখান থেকে ইন্ডেন্টেশন শুরু হয় সেখানেই শেষ হয়।
* শূন্য নয় এমন ভ্যালুকে পাইথন True হিসাবে গণ্য করে। None এবং 0 কে False হিসাবে গণ্য করে।

### 

### if স্টেটমেন্টের ফ্লোচার্ট



### ****উদাহরণঃ**** পাইথন if স্টেটমেন্ট

# ধনাত্বক সংখ্যা হলে আমরা একটি ম্যাসেজ আউটপুট পাবো

num = 5

if num > 0:

print(num, "is a positive number.")

print("This is always printed.")

num = -5

if num > 0:

print(num, "is a positive number.")

print("This is also always printed.")

**আউটপুট**

5 is a positive number.

This is always printed.

This is also always printed.

**উদাহরণের ব্যাখ্যা**

* উপরের উদাহরণে num > 0 হচ্ছে testExpression।
* যদি testExpression এর ভ্যালু True হয় তাহলে if এর বডি সম্পাদিত হবে।
* যখন ভ্যারিয়েবল num এর ভ্যালু 5 এর সমান হয় তখন testExpression এর ভ্যালু true এবং if এর ভেতরে অবস্থিত বডি সম্পাদিত হবে।
* যদি ভ্যারিয়েবল num এর ভ্যালু -5 এর সমান হয় তাহলে testExpression এর মান false হয় এবং if এর ভেতরে অবস্থিত বডিকে স্কিপ করে পরবর্তী স্টেটমেন্ট সম্পাদিত হয়।
* print() স্টেটমেন্ট if ব্লক এর বাহিরে থাকায় এটি testExpression কে গ্রাহ না করে সব সময়ই সম্পাদিত হয়।

**মাল্টিপল বা নেস্টেড কন্ডিশন**

**Nested if**

কোন একটা if কন্ডিশনের ভিতর নেস্টেড (nested) ভাবে আরো এক বা একাধিক কন্ডিশনাল স্টেটমেন্ট ব্যবহার করা যায়, এটাকে নেস্টেড কন্ডিশন বলে। একটা উদাহরণ দেখে নেওয়া যাক –

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | # This program is an example of nested conditions in python    weather=25    if(weather<20):      print("The weather is cold.")  else:      if (weather>=20 and weather<=27):          print("The weather is temperate.")      elif(weather>27):          print("The weather is hot.") |

ফলাফল-

The weather is temperate.

এখানে পূর্বের উদাহরণের মতই কাজ হচ্ছে কিন্তু প্রথমেই আমরা যাচাই করে নিচ্ছি যে এটা ঠান্ডা আবহাওয়া কিনা, যদি না হয় তখন আমরা else এর ভিতর যেয়ে যাচাই করে নিচ্ছি যে এটা নাতিশীতোষ্ণ নাকি গরম।

এখানে else এর পরে যে if এবং elif ব্যবহার করা হয়েছে এটা নেস্টেড কন্ডিশন।

#### লুপ পরিচিতি

for লুপ দিয়ে খুব সহজেই যেকোনো সিকোয়েন্স টাইপ অবজেক্ট যেমন list, string ইত্যাদির মধ্যে iterate করা যায়। তাহলে দেখি উপরের প্রোগ্রামটি কিভাবে ফর লুপ ব্যবহার করে করা সম্ভব,

1

fruits = ["Apple", "Orange", "Pineapple", "Grape"]

2

# Lets make juice with these fruits

3

4

for fruit in fruits:

5

print(fruit + " Juice!")

Copied!

আউটপুট,

1

Apple Juice!

2

Orange Juice!

3

Pineapple Juice!

4

Grape Juice!

Copied!

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

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

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

1

for i in range(10):

2

print(i)

Copied!

আউটপুট,

1

0

2

1

3

2

4

3

5

4

6

5

7

6

8

7

9

8

10

9

Copied!

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

এখন পর্যন্ত range ফাংশন এ প্যারামিটার হিসবে আমরা একটিমাত্র আর্গুমেন্ট দিচ্ছিলাম। আমরা যখন ফর লুপে range(10) লিখেছিলাম, এটি মূলত ০ থেকে ৯ পর্যন্ত প্রিন্ট করেছে। আবার একইভাবে যদি range(20) বসাই, তাহলেও একইভাবে ০ থেকে ১৯ পর্যন্ত প্রিন্ট করে দেখাতো । কিন্তু আমরা চাইলে আমাদের রেঞ্জ পছন্দমত বলে দিতে পারি, যেমন ধরুন, ৫ থেকে শুরু করে ৯ পর্যন্ত। এজন্য আমাদের যা করতে হবে , রেঞ্জ ফাংশনে দুইটা মান বা প্যারামিটার পাস করতে হবে, প্রথমটা শুরু এবং পরেরটা শেষের মান। নিচের কোডটা খেয়াল করি -

1

# start with 5 and ends with 10

2

for i in range(5, 10):

3

print(i)

Copied!

আউটপুট

1

5, 6, 7, 8, 9

Copied!

এছাড়াও আমরা স্টেপ সাইজও বলে দিতে পারি এখানে। মানে, কত ঘর পরপর মান বা ভ্যালু প্রিন্ট করবে সেটা। এরজন্য আমাদের উপরের মানদুটির সাথে আরো একটি প্যারামিটার দিতে হবে, যেটা হবে ইন্টারভাল বা স্টেপ সাইজ। যেমন ধরুন, আমরা ৫ থেকে ১৫ পর্যন্ত প্রিন্ট করবো এবং চাই যে তিন ঘর পরপর প্রিন্ট হোক। তাহলে এর জন্য আমাদের রেঞ্জ ফাংশনের ভিতরে যথাক্রামে, 'শুরু, শেষ, স্টেপ\_সাইজ' মানগুলো বসাতে হবে। নিচের কোডটা খেয়াল করি -

1

# start with 5 and ends with 15 and step size 3

2

for i in range(5, 15, 3):

3

print(i)

Copied!

আউটপুট

1

5 , 8 , 11 , 14

Copied!

আচ্ছা এ পর্যন্ত বুঝা গেলে আমরা একটি কাজ করতে পারি, এ পর্যন্ত তো আমরা সামনের দিকে ভ্যালু প্রিন্ট করা দেখলাম, কেননা আমরা এবার রেঞ্জ ব্যবহার করে এমন একটি কোড লিখি যেটা উল্টো দিকে ভ্যালু প্রিন্ট করবে, মানে ধরুন ১০ থেকে শুরুে হয়ে ০ পর্যন্ত যাবে এবং চলুন এর সাথে স্টেপ সাইজও ব্যবহার করে ফেলি, যেনো দুই ঘর পিছিয়ে পিছিয়ে ভ্যালু প্রিন্ট করে। নিচের কোডটা দেখলে বিষয়টি আরো পরিষ্কার হবে -

1

# start with 10

2

# end with 0

3

# step size -2

4

for i in range(10, 0, -2):

5

print(i)

Copied!

আঊটপুট

1

10

2

8

3

6

4

4

5

2

Copied!

আচ্ছা একটি মজার বিষয় জেনে রাখি, এই পর্যন্ত রেঞ্জ নিয়ে কাজ করলে আপনি হয়তো খেয়াল করেছেন , রেঞ্জ ফাংশনে আপনি শুধু ইন্টিজার ভ্যালুই দিতে পারেন, ফ্লোট টাইপ ভ্যালু দিতে পারেন না। সত্যি বলতে রেঞ্জ ফাংশন ফ্লোট টাইপ ভ্যালু আর্গুমেন্ট হিসেবে নেয় না।

কিন্তু আমরা চাইলে এর জন্য কাস্টম ফাংশন তৈরি করে নিতে পারি। এখানে আমরা একটি ইউজার ডিফাইন ফাংশন তৈরি করবো, যেটা রেঞ্জ ফাংশনের মতনই কাজ করবে, কিন্তু পার্থক্য হচ্ছে, এটি ফ্লোট টাইপ ডাটা নিয়েও কাজ করতে পারবে। আচ্ছা, ফাংশন সেকশনে ইউজার ডিফাইন ফাংশন নিয়ে বিস্তারিত ব্যাখ্যা আছে। আপনি যদি এর সাথে পরিচিত না হয়ে থাকেন, তাহলে আগে সে বিষয়টি সম্পর্কে জেনে আসুন, তারপর এটি পড়ুন, এতে আপনার বুঝতে সুবিধা হবে। আচ্ছা, নিচের কোডটা খেয়াল করি -

1

# we need all of those three argumet, such as start, stop, step

2

def frange(start, stop, step):

3

i = start

4

while i < stop:

5

yield i

6

i += step

7

for i in frange(0.5, 1.0 ,0.1):

8

print(i)

Copied!

আউটপুট

1

0.5

2

0.6

3

0.7

4

0.7999999999999999

5

0.8999999999999999

6

0.9999999999999999

Copied!

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

1

for letter in 'Python': # Here "Python" acts like a list of characters

2

print(letter)

Copied!

আউটপুট,

1

P

2

y

3

t

4

h

5

o

6

n

Copied!

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

1

for i in range(20):

2

if i == 5:

3

continue

4

if i > 9:

5

break

6

print(i)

7

8

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

Copied!

1

0

2

1

3

2

4

3

5

4

6

6

7

7

8

8

9

9

10

Printed first 10 numbers except 5!

Copied!

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

#### লুপ কভাবে স্টপ করবো ?

# **লুপ মানে ফাঁস। ফাঁস আমরা সবাই চিনি। এটা নিয়ে রাজনীতি করার কিছু নাই। কিন্তু প্রোগ্রামিংয়ের লুপ নিয়ে বলার আছে অনেক কিছুই। এই লুপ মানে হল বারবার। যখন একই কাজ বারবার করার দরকার পড়ে তখন লুপ ব্যবহার করতে হয়। লুপকে কন্ডিশনাল লজিকের ভিতরেই ফালানো যায়। কারণ লুপে কন্ডিশনের উপর ভিত্তি করেই একই কাজ বারবার করা হয়।**

আমরা একটা সমস্যার কথা চিন্তা করি। আমরা তো প্রিন্ট করতে সবাই পারি। আমাদের বলা হল, ১ থেকে ১০ পর্যন্ত সংখ্যাগুলো প্রিন্ট করতে হবে।

break ও continue ব্যবহার করে আমরা একটি লুপ স্টপ করতে পারি।

#### হোয়াইল লুপ

# **পাইথন while লুপ**

কোনো নির্দিষ্ট কোড ব্লককে রিপিট করার জন্য প্রোগ্রামিং এ লুপ ব্যবহৃত হয়। এই অধ্যায়ে আপনি পাইথনে while লুপ তৈরি করা শিখবেন।

যতক্ষণ পর্যন্ত testExpression বা কন্ডিশন true থাকে ততক্ষণ পর্যন্ত কোড ব্লককে ইটারেট করার জন্য পাইথনে while লুপ ব্যবহৃত হয়।

কোনো কোড ব্লককে কত সংখ্যকবার ইটারেট(iterate) করতে হবে তা পূর্বে থেকে জানা না থাকলে আমরা সাধারণত লুপ ব্যবহার করি।

## **পাইথন while লুপ এর সিনট্যাক্স**

while testExpression:

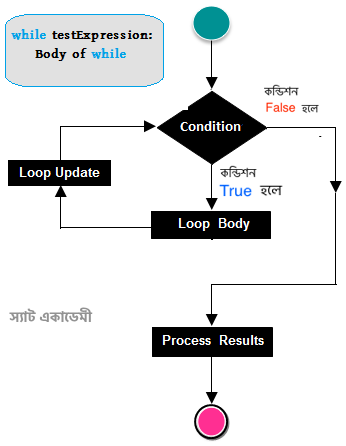
Body of while

**Syntax এর ব্যাখ্যা**

* while লুপে প্রথমে testExpression কে চেক করে। প্রোগ্রাম লুপের Body এর মধ্যে কেবল তখনই প্রবেশ করে( লুপের কোড ব্লক রান করে) যখন testEexpression এর মান True হয়।
* একবার ইটারেশন(iteration) এর পরে testExpression পুনরায় চেক হয়। testExpression এর মান False না হওয়া পর্যন্ত এই প্রক্রিয়া চলতেই থাকে।
* পাইথনে while লুপের body নির্ণয়ের জন্য ইন্ডেন্টেশন ব্যবহৃত হয়।
* Body শুরু হয় ইন্ডেন্টেশন দিয়ে এবং শেষ হয় যেখান থেকে প্রথম ইন্ডেন্টেশন শুরু হয়েছিল।

**নোটঃ** শূন্য নয় এমন যেকোনো ভ্যালুকে পাইথন True হিসাবে গণ্য করে। None এবং 0 কে False হিসাবে গণ্য করে।

## **while লুপ ফ্লোচার্ট**



**উদাহরনঃ পাইথন while লুপ**

# স্বাভাবিক সংখ্যার যোগফল নির্ণয়ের প্রোগ্রাম

# sum = 1+2+3+...+n

# ইউজার থেকে ইনপুট গ্রহন এর জন্য

# কমেন্ট তুলে দিন

# n = int(input("Enter n: "))

n = 10

# count এবং sum কে ইনিশিয়ালাইজ করা

count = 1

sum = 0

while count <= n:

sum = sum + count

count = count + 1 # update counter

#যোগফল প্রিন্ট হবে

print("The sum is", sum)

**আউটপুট**

Enter n: 10

The sum is 55

**উদাহরণের ব্যাখ্যা**

* উপরের প্রোগ্রামে testExpression ততক্ষণ পর্যন্ত True থাকবে যতক্ষণ পর্যন্ত ভ্যারিয়েবল count এর মান 10 এর সমান বা ছোট থাকবে। আমাদের প্রোগ্রামে count এর মান 10।
* লুপের body এর মধ্যে কাউন্টার(counter) ভায়রিয়েবলের মান বৃদ্ধি করা দরকার। এটি অত্যন্ত জরুরী, মজার বিষয় হলো প্রায় সময়ই আমরা এটা করতে ভুলে যায়। এটা করতে ভুলে গেলে প্রোগ্রাম অসীম সংখ্যকবার চলতে থাকবে, অর্থাৎ শেষই হবে না।
* পরিশেষে ফলাফল প্রদর্শিত হবে।

## 

## **else সহ while লুপ**

for লুপের ন্যায় while লুপেরও অতিরিক্ত else ব্লক থাকতে পারে।

এক্ষেত্রে while লুপের কন্ডিশন False হলে else অংশ সম্পাদিত হয়। while লুপ বন্ধ করার জন্য break স্টেটমেন্ট ব্যবহার করা হয়। এটি else অংশকেও এড়িয়ে যায়।

সুতরাং লুপের else অংশ কেবল তখনই সম্পন্ন হয় যখন কোনো break স্টেটমেন্ট থাকে না এবং কডিশন False হয়।

**নিচের উদাহরণের মাধ্যমে এটি আপনার কাছে আরও পরিষ্কার হয়ে যাবে।**

# for লুপ এর সাথে else স্টেটমেন্ট এর ব্যবহার

counter = 0

while counter < 5:

print("Inside while loop")

counter = counter + 1

else:

print("Inside else ")

**আউটপুট**

Inside while loop

Inside while loop

Inside while loop

Inside while loop

Inside while loop

Inside else

**উদাহরণের ব্যাখ্যা**

* উপরের প্রোগ্রামে স্ট্রিং Inside while loop কে ৫ বার প্রিন্ট করার জন্য আমরা কাউন্টার ভ্যারিয়েবল ব্যবহার করেছি।
* while লুপের ৬ষ্ট ইটারেশনে কন্ডিশন False হয়। সুতরাং else অংশ সম্পাদিত(executed) হয়।

#### এরর হ্যান্ডলিং: প্রাথমিক ধারণা

# **এরর হ্যান্ডলিং**

ইতিপূর্বে আমরা দেখে এসেছি নানা সময়ে নানা কারণে পাইথন নানা প্রকার এরর থ্রো করতেছে। ঐসব এরর আমাদের জিন্দেগী তামা-তামা কইরা দিছে। এখন আমরা ঐসব এরর হ্যান্ডেল করার জাদুমন্ত্র শিখব। তবে তার আগে এক নজরে আমরা গুরুত্বপূর্ণ স্টান্ডার্ড এক্সেপশনগুলো দেখে নেব। আর দেখব কখন তারা রেইজ (raise) হয়।

| এক্সেপশন | বর্ণনা |
| --- | --- |
| FileNotFoundError | যখন কোন নির্দিষ্ট ফাইলের অস্তিত্ব খুঁজে পাওয়া যায় না তখন রেইজ হয়। |
| ImportError | যখন import স্টেটমেন্ট ফেল করে তখন রেইজ হয়। |
| IndentationError | যখন কোডব্লকে ইনডেন্টেশন মেইনটেইন করা হয় না ঠিকমত তখন রেইজ হয়। |
| IndexError | যখন কোন সিকুয়েন্সে কোন নির্দিষ্ট ইনডেক্স খুঁজে পাওয়া যায় না তখন রেইজ হয়। |
| KeyboardInterrupt | যখন **Ctrl + C** চেপে ইউজার কোন প্রোগ্রামের এক্সিকিউশন থামিয়ে দেয় তখন রেইজ হয়। |
| KeyError | যখন ডিকশনারিতে কোন নির্দিষ্ট কী খুঁজে পাওয়া যায় না তখন রেইজ হয়। |
| NameError | যখন নির্দিষ্ট কোন লোকাল বা গ্লোবাল ভ্যারিয়েবল খুঁজে পাওয়া যায় না তখন রেইজ হয়। |
| OSError | যখন কোন সিস্টেম ফাংশন সিস্টেম রিলেটেড কোন এরর রিটার্ন করে তখন রেইজ হয়। |
| RuntimeError | যখন কোন এরর হ্যান্ডেল করার জন্য জেনারেল কোন স্টান্ডার্ড এক্সেপশন পাওয়া যায় না তখন রেইজ হয়। |
| StopIteration | যখন next() মেথড আর কোন অবজেক্টকে পয়েন্ট করে না তখন রেইজ হয়। |
| SyntaxError | যখন সিনট্যাক্সে ঘাপলা থাকে তখন রেইজ হয়। |
| SystemError | যখন পাইথন ইন্টারপ্রিটার কোন ইন্টারনাল এরর খুঁজে পায় তখন রেইজ হয়। |
| TabError | যখন কোডে একই সাথে ট্যাব আর স্পেস দিয়ে ইন্ডেন্টেশন করা হয় তখন রেইজ হয়। |
| TypeError | যখন কোন ফাংশন বা অপারেশন কোন অসঙ্গত টাইপের অবজেক্টে অ্যাপ্লাই করা হয় তখন রেইজ হয়। |
| UnboundLocalError | এটা হল NameError এর শিষ্য। যখন কোন ফাংশন বা মেথডে কোন লোকাল ভ্যারিয়েবল ব্যবহার করা হয় কিন্তু ঐ ভ্যারিয়েবলের কোন ভ্যালু থাকে না তখন রেইজ হয়। |
| ValueError | যখন কোন বিল্ট-ইন অপারেশন বা ফাংশন এমন কোন আর্গুমেন্ট গ্রহণ করে যার টাইপ ঠিক থাকলেও ভ্যালুতে গন্ডগোল এবং IndexError দিয়ে এই সমস্যা ধামাচাপা দেয়া যায় না তখন রেইজ হয়। |
| WindowsError | উইন্ডোজ অপারেটিং সিস্টেম ঘটিত সমস্যায় রেইজ হয়। |
| ZeroDivisionError | যখন কোন ভ্যালুকে শুন্য দিয়ে ভাগ করার চেষ্টা করা হয় তখন রেইজ হয়। |

## **এক্সেপশন (Exception) কি?**

সহজ কথায়, এক্সেপশন হল একটা ঘটনা যা প্রোগ্রাম চলার সময় কোন সমস্যার উদগীরণ হলে ঘটে। একটা উদাহরণ দিয়ে বিষয়টা আরো পরিষ্কার করে তুলে ধরা যাক। ফাইল চাপ্টারে আমরা সবার শেষে যে প্রোগ্রামটা লিখেছিলাম সেটা আবার চালাব আমরা। তবে এবার test.txt ফাইলটা ঐ জায়গা থেকে ডিলিট করে দেয়ার পর।

with open('test.txt', 'r') as my\_file:

content = my\_file.read()

print(content)

**আউটপুট**

Traceback (most recent call last):

File "/home/ugcoder/Desktop/test.py", line 1, in <module>

with open('test.txt', 'r') as my\_file:

FileNotFoundError: [Errno 2] No such file or directory: 'test.txt'

test.txt ফাইলটা না পেয়ে FileNotFoundError থ্রো করেছে পাইথন, তারপর প্রোগ্রামের এক্সিকিউশন বন্ধ হয়ে গেছে। ফলে শেষের নির্ভেজাল স্টেটমেন্টটাও এক্সিকিউট হয় নাই। এইরকম সমস্যার নিরসন করতেই এক্সেপশন হ্যান্ডলিং বা এরর হ্যান্ডলিং।

## **try … except**

উপরের এররটাই এবার আমরা হ্যান্ডেল করব। আগে আমরা প্রোগ্রামটা দেখব তারপর ব্যাখ্যায় যাব।

try:

with open('test.txt', 'r') as my\_file:

content = my\_file.read()

print(content)

except:

print('The file does not exist.')

print('Made by Maateen.')

**আউটপুট**

The file does not exist.

Made by Maateen.

এবার কিন্তু আর প্রোগ্রামটা ঠুস করেই শেষ হয়ে যায়নি। এররটা হ্যান্ডেল হয়ে শেষ স্টেটমেন্ট অবধি সুন্দরভাবে প্রোগ্রাম এক্সিকিউট হয়েছে।

try...except এর ব্যাপারটা হল এরকম: নরমাল কোডগুলো try ব্লকের ভিতর থাকবে। আর কোন এক্সেপশন রেইজ হলে except ব্লকের কোড এক্সিকিউট হবে। কোন নির্দিষ্ট এক্সেপশন হ্যান্ডেল করার জন্য কোড লিখতে চাইলে except এর পরে স্পেস দিয়ে এক্সেপশনের নাম দিয়ে দিতে হয়। আবার চাইলে এক্সেপশনের থ্রো করা মেসেজ হোল্ড করে প্রিন্ট করে ইউজারকে দেখানোও যায়। একটা উদাহরণ দেখা যাক।

try:

with open('test.txt', 'r') as my\_file:

content = my\_file.read()

print(content)

except FileNotFoundError:

print('The file does not exist.')

print('Made by Maateen.')

try:

my\_list = []

print(my\_list[0])

except IndexError as e:

print(e)

**আউটপুট**

The file does not exist.

Made by Maateen.

list index out of range

এখানে আমরা এক্সেপশনের নাম ধরে এরর হ্যান্ডেল করেছি। শেষের এক্সেপশনে পাইথনের জেনারেট করা এরর মেসেজই প্রিন্ট করে দিয়েছি। আচ্ছা, একটা try ব্লকের কোডে কয়েকটা এরর হতে পারে। সেক্ষেত্রে কিভাবে এরর হ্যান্ডেল করব আমরা?

আসলে একটা try ব্লকের জন্য যত খুশি তত except ব্লক লেখা যায়। তবে প্রতিটা except ব্লকে এক্সেপশনের নাম উল্লেখ করতে হবে। একটা নির্দিষ্ট এক্সেপশন রেইজ হলেই কেবল ঐ নির্দিষ্ট except ব্লক এক্সিকিউট হবে। একটা উদাহরণ দেখা যাক।

try:

my\_file = open('test.txt')

content = my\_file.read()

i = int(content.strip())

except IOError as e:

errno, strerror = e.args

print("I/O error({0}): {1}".format(errno,strerror))

except ValueError:

print("No valid integer in line.")

except:

print("Unexpected error!")

**আউটপুট**

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

এখানে আমরা একটা try ব্লকের জন্য তিনটা except ব্লক লিখেছি। প্রথমটা IOError এর জন্য, দ্বিতীয়টা ValueError এর জন্য আর শেষেরটা ঐ দুইটা বাদে যেকোন এররের জন্য। এ ধরনের except কে পাইথনিক ভাষায় Bare Except বলা হয়। বাংলায় অবশ্য আমরা এর একটা নাম দিতে পারি - ল্যাংটা এক্সেপ্ট। আর পারতপক্ষে ল্যাংটা এক্সেপ্ট স্কিপ করা উচিত আমাদের। কারণ, বেয়ার এক্সেপ্ট সব ধরনের এররকে হাইড করে দেয় ফলে আমরা জানতেই পারব না ঠিক কোন এক্সেপশনটাকে আমরা ক্যাচ করছি। তাই দেশ ও জাতির বৃহত্তর কল্যাণের স্বার্থে আমরা সবসময় এক্সেপশনের নাম উল্লেখ করে এরর হ্যান্ডেল করব।

আমরা চাইলে সবগুলো এররকে একটা except ব্লকে সেটেল করে দিতে পারতাম।

try:

my\_file = open('test.txt')

content = my\_file.read()

i = int(content.strip())

except (IOError, ValueError):

pass

## **try … except … else**

else ব্লক except ব্লকের শেষে বসে। try ব্লকে কোন এক্সেপশন রেইজ না হলেই কেবল else ব্লকের কোড এক্সিকিউট হয়। একটা উদাহরণ দেখা যাক।

try:

a = 5

b = 8

print(a + b)

except ValueError as e:

print(e)

else:

print('There is no exception.')

**আউটপুট**

13

There is no exception.

কোন এক্সেপশন রেইজ হয়নি। তাই else ব্লকের কোড এক্সিকিউট হয়েছে।

## **try … except … finally**

finally ব্লক একেবারে শেষে বসে। আর কোন এক্সেপশন রেইজ হোক আর নাই বা হোক, finally ব্লকের কোড ঠিকই এক্সিকিউট হয়। এজন্য একে ক্লীন-আপ অ্যাকশন বলা হয়। একটা উদাহরণ দেখা যাক।

try:

with open('test.txt', 'r') as my\_file:

content = my\_file.read()

print(content)

except FileNotFoundError:

print('The file does not exist.')

finally:

print('To be or not to be that is the question.')

**আউটপুট**

The file does not exist.

To be or not to be that is the question.

test.txt ফাইলটা না থাকায় FileNotFoundError এক্সেপশন রেইজ হয়েছে। কিন্তু তারপরও finally ব্লকের কোড ঠিকই রেইজ হয়েছে।

#### এরর হ্যান্ডলিং এর কিছু নিয়ম

## **রেইজ এক্সেপশন**

পাইথনে তো অনেক বিল্ট-ইন এক্সেপশন আছে ( <https://docs.python.org/3/library/exceptions.html> )। আমরা চাইলে এদেরকে নিজ থেকে রেইজ করতে পারি।

try:

raise NameError('Hey! It is a custom error message.')

except NameError as e:

print(e)

**আউটপুট**

Hey! It is a custom error message.

এক্সেপশন রেইজ করার জন্য raise স্টেটমেন্ট ব্যবহার করতে হয়। raise এর পরে বিল্ট-ইন এক্সেপশনের নাম দিয়ে এর সাথেই ব্রাকেটের ভিতর এরর মেসেজ স্ট্রিং হিসাবে পাস করতে হয়।

বিল্ট-ইন এক্সেপশন ছাড়াও পাইথনে ইউজার-ডিফাইন্ড এক্সেশন ব্যবহার করা যায়। কিন্তু আমরা তো এখনও ছোট। তাই এখনই আমরা এটা শিখব না। ক্লাস, অবজেক্ট, মেথড সম্পর্কে যখন আমাদের ধারণা পরিষ্কার হবে তখন আমরা ইউজার-ডিফাইন্ড এক্সেপশন পয়দা করব। ততক্ষণ হ্যাপি এরর হ্যান্ডলিং!

#### ফাঙ্শন পরিচিতি

# **ফাংশন**

ফাংশনের সঙ্গে পরিচয় আমাদের অনেক আগেই হয়ে গিয়েছে। প্রথমেই যখন Hello world প্রিন্ট করেছিলাম, তখনই আমরা print() ফাংশন ব্যবহার করেছি। এছাড়াও আরো অনেক ফাংশন ব্যবহার করেছি, যেমন input(), len(), type() ইত্যাদি। আবার আমরা turtle মডিউল যখন ব্যবহার করেছি, তখন টার্টলের অনেক ফাংশনও ব্যবহার করেছি, যেমন forward(), left(), right(), dot() ইত্যাদি। ফাংশনগুলো কী কাজ করে, এটি আমাদের জানতে হবে, কিন্তু কিভাবে কাজ করে, সেটি এখন আমাদের জানার প্রয়োজন নেই। এসব ফাংশন তৈরি করে দেওয়া না থাকলে আমাদেরকে অনেক বেশি কষ্ট করতে হতো। যেমন print() ফাংশনটি কিভাবে কাজ করবে, সেটি পাইথনের বিল্টইনস্ মডিউলে লেখা আছে (তবে এই মডিউলটি আমাদের ইমপোর্ট করতে হয় না)। তেমনি dot() কিভাবে কাজ করবে, সেটি লেখা আছে টার্টল মডিউলের ভেতরে। এই অধ্যায়ে আমরা দেখবো, কিভাবে ফাংশন তৈরি করতে হয়, কিভাবে মডিউল তৈরি করতে হয় এবং প্রয়োজনীয় কিছু ফাংশনের ব্যবহার।

আমরা কখন ফাংশন তৈরি করবো? যখন আমাদেরকে একটি নির্দিষ্ট কাজ করতে হবে এবং সেই কাজটি একাধিকবার করার প্রয়োজন হবে, তখন আমরা সেই কাজের জন্য একটি ফাংশন তৈরি করে ফেলবো। আবার কখনও কখনও একটি কাজ একবার করলেও আমরা সেটির জন্য আলাদা ফাংশন তৈরি করতে পারি, যেন কোড বুঝতে সহজ হয়। যেমন ধরা যাক, আমাকে কোনো প্রোগ্রামে অনেকবার দুটি সংখ্যা যোগ করতে হবে। তখন আমরা এই যোগ করার জন্য একটি ফাংশন তৈরি করে ফেলতে পারি এভাবে :

def add(n1, n2):

return n1 + n2

আমরা দেখতে পাচ্ছি যে, ফাংশনের শুরুতে লিখতে হবে def, তাহলে পাইথন বুঝবে যে এখানে একটি ফাংশন তৈরি করা হচ্ছে (বা সংজ্ঞায়িত করা হচ্ছে, definition শব্দের প্রথম তিন অক্ষর def)। তারপর ফাংশনের নাম দিতে হবে। আমরা নাম দিয়েছি add। ফাংশনের নাম দেখে যেন বোঝা যায় যে, ফাংশনটি কী কাজ করবে। তারপরে প্রথম বন্ধনীর ভেতরে ফাংশনের প্যারামিটার লিখতে হবে। সব ফাংশনে প্যারামিটার থাকে না। প্যারামিটার থাকবে কী না এবং কয়টি, সেটি নির্ভর করে আমরা কী ফাংশন তৈরি করছি, তার ওপর। যেমন এখানে আমি তৈরি করছি দুটি সংখ্যার যোগফল বের করার ফাংশন। তাহলে তো আমাকে কোন দুটি সংখ্যা আমি যোগ করবো, সেগুলো জানতে হবে বা ইনপুট নিতে হবে। ফাংশনের ক্ষেত্রে প্যারামিটার হচ্ছে ইনপুট নেওয়ার পদ্ধতি। তারপরে একটি কোলন চিহ্ন দিতে হবে। পরের লাইন থেকে ফাংশনের ভেতরের কোড লিখতে হবে এবং সেগুলো ইনডেনটেশন করা থাকতে হবে, নইলে পাইথনের পক্ষে বোঝা সম্ভব হবে না যে কোন অংশ ফাংশনের ভেতরে আর কোন অংশ বাইরে। ফাংশন থেকে আবার এক বা একাধিক জিনিস ফেরত পাঠানো যায়, যাকে বলে রিটার্ন করা। আমাদের যেমন যোগফল ফেরত পাঠানো দরকার, তাই আমরা n1 + n2 এর মান রিটার্ন করছি। আমরা এখন ফাংশনটির ব্যবহার দেখবো। আমাদের কোনো নিয়মকানুন মুখস্থ করার প্রয়োজন নেই, চর্চা করে ও চিন্তা করে আমরা প্রোগ্রামিং শিখবো।

>>> def add(n1, n2):

... return n1 + n2

...

>>> n = 10

>>> m = 5

>>> result = add(n, m)

>>> print(result)

15

>>>

>>> number1 = 10

>>> number2 = 10

>>> result = add(number1, number2)

>>> print(result)

20

>>>

>>> n1 = 20

>>> n2 = 10

>>> print(add(n1, n2))

30

>>> print(add(2.5, 3.5))

6.0

>>>

#### ফাঙ্শন এর পেরামিটার ও আরগুমেন্ট

**ফাংশন আর্গুমেন্ট**

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

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

1

def show\_double(x):

2

print(x\*2)

3

​

4

show\_double(2)

5

show\_double(100)

আউটপুট,

1

4

2

200

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

আর্গুমেন্টকে ফাংশনের দুটি প্রথম বন্ধনীর মধ্যে ডিফাইন করতে হয়।

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

1

def make\_sum(x, y):

2

z = x + y

3

print(z)

4

​

5

make\_sum(5, 10)

6

make\_sum(500, 500)

আউটপুট,

1

15

2

1000

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

1

def make\_sum(x, y):

2

z = x + y

3

print(z)

4

​

5

make\_sum(5, 10)

6

print(z)

আউটপুট,

1

15

2

...

3

NameError: name 'z' is not defined

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

**মাল্টিপল প্যারামিটার হ্যান্ডলিং | আর্বিটরারি আর্গুমেন্ট লিস্ট**

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

1

def make\_sum(\*args):

2

sum = 0

3

for num in args: # Here, args is like a Tuple which is (10, 20, 30, 40)

4

sum += num

5

return sum

6

​

7

print(make\_sum(10, 20, 30, 40))

**আউটপুট**

1

100

# **পাইথনে \* এর অর্থ**

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

পাইথনে \*\* এর অর্থ

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

1

def print\_dict(\*args):

2

print (args)

3

​

4

​

5

print\_dict(a=1, b=2)

আউটপুট,

1

TypeError Traceback (most recent call last)

2

<ipython-input-2-9970453fce76> in <module>()

3

----> 1 print\_dict(a=1, b=2)

4

​

5

TypeError: print\_dict() got an unexpected keyword argument 'a'

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

1

def print\_dict(\*\*kwargs):

2

print(kwargs)

3

​

4

​

5

print\_dict(a=1, b=2, c=3)

আউটপুট,

1

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

আমরা যদি কোডটা আরেকটু গুছিয়ে লেখি,

1

def print\_dict(\*\*kwargs):

2

for args in kwargs:

3

print("{0} : {1}".format(args, kwargs[args]))

4

​

5

​

6

print\_dict(a=1, b=2, c=3)

আউটপুট,

1

a : 1

2

c : 3

3

b : 2

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

1

def print\_all(a, \*args, \*\*kwargs):

2

print(a)

3

print(args)

4

print(kwargs)

5

​

6

​

7

print\_all(10, 20, 30, 50, b=5, c=10)

আউটপুট,

1

10

2

(20, 30, 50)

3

{'c': 10, 'b': 5}

**প্যারামিটার ও আর্গুমেন্ট**

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

#### ভ্যারিয়েবল স্কোপ

## **লোকাল ও গ্লোবাল ভেরিয়েবল**

**লোকাল ভেরিয়েবল** : একটি প্রোগ্রামে যেকোন ফাংশনে তৈরিকৃত variable ই **লোকাল ভেরিয়েবল**, এমনকি main function এর ভিতরে যেই variable declare করা হয় তাও **লোকাল ভেরিয়েবল** । একটি প্রোগ্রাম যখন কোনো ফাংশনের ভেতরে ঢুকে কম্পাইল করা শুরু করে , তখন সেই ফাংশনের ভেতরে যেই ভেরিয়েবলগুলো ডিক্লেয়ার করা হয়, সেগুলো মেমোরির এক জায়গায় থাকে এবং যখনই প্রোগ্রাম সেই ফাংশন থেকে বের হয়ে যায়, তখন আর সেই ভেরিয়েবলগুলোর নাম-ঠিকানা প্রোগ্রামের কাছে সংরক্ষিত থাকে না, তাই সেগুলো একসেস করা যায় না। এটিই হলো লোকাল ভেরিয়েবল ।

আর যদি কোন ফাংশনের বাইরে ভেরিয়েবল ডিক্লেয়ার করেন তবে তা হবে গ্লোবাল ভেরিয়েবল। ঐ ভেরিয়েবলকে প্রোগ্রামের সকল ফাংশন ব্যবহার করতে পারে। সুতরাং কোন ভেরিয়েবলকে একাধিক ফাংশনে ব্যবহার করতে চাইলে অবশ্যই তাকে **গ্লোবাল ভেরিয়েবল** হিসেবে ঘোষণা করতে হবে। গ্লোবাল ভেরিয়েবল সাধারণত একটি প্রোগ্রাম যা একটি ফাংশন এর বাইরে সংজ্ঞায়িত করা হয়। **গ্লোবাল ভেরিয়েবল** আপনার প্রোগ্রাম জীবনকাল জুড়ে তাদের মূল্য রাখা হবে এবং তারা প্রোগ্রামের জন্য নির্ধারিত কর্ম কোনো ভিতরে প্রবেশ করা যেতে পারে।

বিশ্বব্যাপী পরিবর্তনশীল কোনো ফাংশন দ্বারা ব্যবহার করা যাবে। এটা বিশ্বব্যাপী একটি পরিবর্তনশীল এর ঘোষণার পর আপনার সম্পূর্ণ প্রোগ্রাম সারা ব্যবহারের জন্য উপলব্ধ করা হয়. বৈশ্বিক এবং স্থানীয় ভেরিয়েবল ব্যবহার করে উদাহরণ অনুসরণ করা হয়:

|  |
| --- |
| #include <stdio.h>/\* global variable declaration \*/ int g;  int main () { /\* local variable declaration \*/ int a, b;  /\* actual initialization \*/ a = 10; b = 20; g = a + b;  printf (“value of a = %d, b = %d and g = %d\n”, a, b, g);  return 0; } |

একটি ফাংশন ভিতরে স্থানীয় এবং গ্লোবাল ভেরিয়েবল কিন্তু স্থানীয় পরিবর্তনশীল এর মান পছন্দের নিতে হবে জন্য একটি প্রোগ্রাম একই নাম থাকতে পারে.নিম্নলিখিত একটি উদাহরণ:

|  |
| --- |
| #include <stdio.h>/\* global variable declaration \*/ int g = 20;  int main () { /\* local variable declaration \*/ int g = 10;  printf (“value of g = %d\n”, g);  return 0; } |

উপরের কোড কম্পাইল এবং মৃত্যুদন্ড কার্যকর করা হয়, এটি নিম্নলিখিত ফলাফল সৃষ্টি করে:

|  |
| --- |
| value of g = 10 |

## **Formal Parameters**

ফাংশন পরামিতি, আনুষ্ঠানিক পরামিতি, যে ফাংশন সঙ্গে স্থানীয় ভেরিয়েবল হিসেবে গণ্য করা হয় এবং তারা গ্লোবাল ভেরিয়েবল উপর পক্ষপাত নিতে হবে.নিম্নলিখিত একটি উদাহরণ:

|  |
| --- |
| #include <stdio.h>/\* global variable declaration \*/ int a = 20;  int main () { /\* local variable declaration in main function \*/ int a = 10; int b = 20; int c = 0;  printf (“value of a in main() = %d\n”, a); c = sum( a, b); printf (“value of c in main() = %d\n”, c);  return 0; }  /\* function to add two integers \*/ int sum(int a, int b) { printf (“value of a in sum() = %d\n”, a); printf (“value of b in sum() = %d\n”, b);  return a + b; } |

উপরের কোড কম্পাইল এবং মৃত্যুদন্ড কার্যকর করা হয়, এটি নিম্নলিখিত ফলাফল সৃষ্টি করে:

|  |
| --- |
| value of a in main() = 10 value of a in sum() = 10 value of b in sum() = 20 value of c in main() = 30 |

## **লোকাল ও গ্লোবাল ভ্যারিয়েবল শুরু করা (initialization)**

একটি স্থানীয় পরিবর্তনশীল সংজ্ঞায়িত করা হয়, এটি সিস্টেমের সক্রিয়া করা হয় না, আপনি এটি নিজের আরম্ভ হবে. গ্লোবাল ভেরিয়েবল অনুসরণ হিসাবে আপনি তাদের সংজ্ঞায়িত যখন সিস্টেম দ্বারা স্বয়ংক্রিয়ভাবে আরম্ভ হয়:

Data Type Initial Default Value

|  |  |
| --- | --- |
| **Data Type** | **Initial Default Value** |
| int | 0 |
| char | ‘\0’ |
| float | 0 |
| double | 0 |
| pointer | NULL |

অন্যথায় সঠিকভাবে, আপনার প্রোগ্রাম ভেরিয়েবল আরম্ভ করার জন্য একটি ভাল প্রোগ্রামিং অভ্যাস অপ্রত্যাশিত ফলাফল উত্পন্ন করতে পারে

### ****লোকাল এবং গ্লোবাল ভেরিয়েবলের মধ্যে পার্থক্য:****

প্রতিটি ভেরিয়েবলের একটি স্কোপ (scope) থাকে। স্কোপ মানে ভেরিয়েবলটি প্রোগ্রামের কোথায় কোথায় পাওয়া যাবে। প্রোগ্রাম যখন কোনো ফাংশনের ভেতরে ঢুকে, তখন সেই ফাংশনের ভেতরে যেই ভেরিয়েবলগুলো ডিক্লেয়ার করা হয়, সেগুলো মেমোরির এক জায়গায় থাকে এবং যখনই প্রোগ্রাম সেই ফাংশন থেকে বের হয়ে যায়, তখন আর সেই ভেরিয়েবলগুলোর নাম-ঠিকানা প্রোগ্রামের কাছে থাকে না, তাই সেগুলো একসেস করা যায় না। তাই ফাংশনের ভেতরের সব ভেরিয়েবল হচ্ছে লোকাল ভেরিয়েবল, এমনকী মেইন ফাংশনের ভেতরে তৈরি করা ভেরিয়েবলও লোকাল ভেরিয়েবল।

এখন মাঝে মাঝে আমাদের এমন ভেরিয়েবল দরকার, যেটা সব ফাংশনের ভেতরে পাওয়া যাবে এবং ফাংশন থেকে বের হয়ে গেলেও ওই ভেরিয়েবলের অস্তিত্ব থাকবে। এইরকম ভেরিয়েবলকে বলে গ্লোবাল ভেরিয়েবল।

#### অবজেক্ট ও ক্লাস পরিচিতি

# **নতুন ক্লাস তৈরি করা**

আমরা যদি নতুন ধরণের অবজেক্ট তৈরি করতে চাই, তাহলে আমাদেরকে প্রথমে নতুন ক্লাস তৈরি করতে হবে। ধরা যাক, আমি একটি কার রেসিং গেমস বানাবো। সেই গেমসের একটি মূল বিষয় হচ্ছে কার (Car) বা গাড়ি। আমাকে অনেক কার অবজেক্ট তৈরি করতে হবে। তাই প্রথমে Car নামে একটি ক্লাস তৈরি করবো। এই ক্লাসে কী কী ডেটা অ্যাট্রিবিউট ও মেথড থাকতে পারে? এখন নিজে নিজে একটি তালিকা তৈরি করার চেষ্টা করে দেখতে হবে। আমি একটি সহজ তালিকা দিচ্ছি। তবে, এটি কোনো পূর্ণাঙ্গ তালিকা নয়, অনেকের মাথায় অন্যকিছুও আসতে পারে।

ডেটা অ্যাট্রিবিউট:

* নাম (name)
* উৎপাদনকারী প্রতিষ্ঠান (manufacturer)
* রং (color)
* তৈরির সাল (year)
* ইঞ্জিনের ক্ষমতা (সিসি) (cc)

মেথড :

* ইঞ্জিন চালু করা (start)
* ব্রেক করা (brake)
* চালানো (drive)
* ডানে-বাঁয়ে ঘোরা (turn)
* গিয়ার পরিবর্তন (change gear)

এভাবে প্রথমে আমাদের ক্লাসের ডিজাইন করতে হবে। তারপর ক্লাস ডায়াগ্রাম (class diagram) তৈরি করতে হবে, তবে এই বইতে সেটি আমি দেখাবো না, তাই আপাতত এটি না শিখলেও চলবে। তারপরে আমরা পাইথনে ক্লাসটি তৈরি করবো। এখন প্রশ্ন হচ্ছে, আমরা কি করে বুঝবো যে আমাদেরকে একটি ক্লাস তৈরি করতে হবে? আমরা যখন সফটওয়্যার তৈরি করবো, এটি যেই সমস্যার সমাধানের জন্য তৈরি, সেটি নিয়ে চিন্তা করে বিশেষ্য পদগুলো (Noun) খুঁজে বের করতে হবে। দেখা যাবে এগুলোর বেশিরভাগের জন্যই আমাদের ক্লাস তৈরি করার দরকার হবে। যেমন, কার রেসিং গেমের ক্ষেত্রে, Car, Player, Track ইত্যাদি। আবার আমরা যদি স্কুল ম্যানেজমেন্ট করার জন্য একটি সফটওয়্যার তৈরি করি, তাহলে সেখানে Student, Teacher, Employee, ClassRoom, Subject, Result ইত্যাদি ক্লাস থাকতে পারে। সবগুলোই কিন্তু বিশেষ্য পদ, অর্থাৎ Noun।

পাইথনে ক্লাস তৈরি করতে হলে প্রথমে class লিখে একটি স্পেস দিয়ে ক্লাসের নাম লিখতে হয়, তারপরে সেই ক্লাসের ভেতরে বিভিন্ন স্টেটমেন্ট লেখা হয়। এখানেও ইনডেন্টেশন করতে হয়। ক্লাস ডিক্লেয়ার করার পরের লাইন থেকে যতটুকু কোড ক্লাসের ভেতরে থাকবে, তারা এক ট্যাব (tab) ডান দিক থেকে শুরু হবে।

class ClassName:

<statement-1>

.

.

.

<statement-N>

ওপরে ClassName হচ্ছে ক্লাসের নাম। আমরা যেই নামের ক্লাস তৈরি করতে চাই, সেই নামটি সেখানে ব্যবহার করবো। আর স্টেটমেন্ট-এর জায়গায় বিভিন্ন ভ্যারিয়েবলে মান রাখা কিংবা মেথড তৈরি করা যাবে।

class Car:

name = "Premio"

color = "white"

def start():

print("Starting the engine")

ওপরে আমরা Car নামে একটি ক্লাস তৈরি করলাম। তারপরে সেখানে name ও color নামে দুটি ডেটা অ্যাট্রিবিউট রাখলাম, আর start নামে একটি মেথড তৈরি করলাম। এখন আমরা এই অ্যাট্রিবিউটগুলো ব্যবহার করতে পারবো। নিচের কোডে সেটি দেখানো হয়েছে :

class Car:

name = "Premio"

color = "white"

def start():

print("Starting the engine")

print("Name of the car:", Car.name)

print("Color:", Car.color)

Car.start()

প্রোগ্রামটি সেভ করে রান করলে নিচের মতো আউটপুট আসবে :

$ python car.py

Name of the car: Premio

Color: white

Starting the engine

[নোট: কার রেসিং গেম তৈরি করার সময় কিন্তু start() মেথড কেবল একটি লাইন প্রিন্ট করবে না, বরং গাড়ি চালু করার একটা শব্দ হবে, অ্যানিমেশন হবে। কিন্তু সেগুলো আমার এই বইতে আলোচনার বিষয় নয়। আমরা বরং অবজেক্ট ওরিয়েন্টেড প্রোগ্রামিং শেখার চেষ্টা করছি।]

এখন কেউ প্রশ্ন করতে পারে যে, সব গাড়ির রং তো সাদা হবে না, আর সব গাড়ির নামও প্রিমিও (Premio) হবে না। তাহলে কী করা যায়? আমরা ক্লাসের ভেতরে ওই ডেটা অ্যাট্রিবিউটগুলোয় ফাঁকা স্ট্রিং অ্যাসাইন করতে পারি। তারপরে সেগুলো পরিবর্তন করা যাবে।

class Car:

name = ""

color = ""

def start():

print("Starting the engine")

Car.name = "Axio"

Car.color = "Black"

print("Name of the car:", Car.name)

print("Color:", Car.color)

Car.start()

এখন প্রোগ্রামটি রান করে দেখব। আর বইটি কিন্তু কেবল পড়ে গেলে চলবে না, সবকিছু বইয়ের সঙ্গে সঙ্গে নিজেও কোড করে প্র্যাকটিস করতে হবে।

$ python car.py

Name of the car: Axio

Color: Black

Starting the engine

আমরা যদি ওপরের কোডে print(dir(Car)) লিখি, তাহলে Car ক্লাসের তিনটি অ্যাট্রিবিউট দেখতে পাবো : ‘color’, ‘name’, ‘start’। সেই সঙ্গে অবশ্য আরো অ্যাট্রিবিউট দেখাবে, যেগুলো \_ দিয়ে শুরু, আপাতত সেগুলো নিয়ে চিন্তা করা দরকার নেই।

এখন পর্যন্ত আমরা সরাসরি Car ক্লাসের বিভিন্ন অ্যাট্রিবিউট ব্যবহার করেছি। এখন আমরা Car ক্লাসের অবজেক্ট তৈরি করবো। এই অবজেক্ট তৈরির কাজটিকে পাইথনের ভাষায় বলে ইনস্ট্যানশিয়েট (instantiate) করা। এজন্য অবজেক্টকে ইনস্ট্যান্সও বলা হয়ে থাকে।

class Car:

name = ""

color = ""

def start():

print("Starting the engine")

# creating a Car object

my\_car = Car()

my\_car.name = "Allion"

print(my\_car.name)

my\_car.start()

এখানে আমি প্রথমে Car ক্লাসের একটি অবজেক্ট তৈরি করলাম : my\_car = Car()। তারপর, my\_car – এর নাম দিয়ে দিলাম : my\_car.name = “Allion” এবং সেটি প্রিন্ট করে দেখলাম। শেষ লাইনে my\_car-কে চালু করার চেষ্টা করলাম, তার start() মেথড কল করে। কিন্তু এখানেই বিপত্তি। প্রোগ্রামটি যদি আমরা চালাই, তাহলে নিচের মতো আউটপুট দেখবো :

$ python car.py

Allion

Traceback (most recent call last):

File "car.py", line 12, in <module>

my\_car.start()

TypeError: start() takes 0 positional arguments but 1 was given

নাম প্রিন্ট হলো ঠিকই, কিন্তু তারপরে এরর দিলো যে, start() takes 0 positional arguments but 1 was given। এর অর্থ হচ্ছে, start() মেথড যখন তৈরি করা হয়েছে, সেখানে সেটি আর্গুমেন্ট হিসেবে কোনো কিছু নেয় না (def start()), মানে 0টি আর্গুমেন্ট নেয়, কিন্তু আমরা 1টি আর্গুমেন্ট পাঠাচ্ছি (but 1 was given)! কিন্তু আমরা কোথায় একটি আর্গুমেন্ট পাঠালাম? আমরা তো লিখেছি my\_car.start()। ব্র্যাকেটের ভেতরে কিছু লিখি নি। আমরা যখন কোনো অবজেক্টের মেথড কল করি (যেমন : my\_car.start()), তখন সেই মেথডের ভেতরে অবজেক্টটি আপনাআপনি চলে যায়। কেন যায়, সেটি আমরা পরবর্তি সময়ে আলোচনা করবো। আমাদের এখন যেটি করতে হবে, সেটি হচ্ছে, def start()-এর বদলে লিখতে হবে def start(self)।

class Car:

name = ""

color = ""

def start(self):

print("Starting the engine")

my\_car = Car()

my\_car.name = "Allion"

print(my\_car.name)

my\_car.start()

এখন প্রোগ্রামটি রান করলে ঠিকঠাক চলবে। self-এর বদলে অন্যকিছু লিখলেও চলে, কিন্তু এই ক্ষেত্রে পাইথনের রীতি হচ্ছে self লেখা।

আমরা যখন একটি Car ক্লাসের অবজেক্ট তৈরি করছি, চাইলে তখনই আমরা সেই অবজেক্টের বিভিন্ন বৈশিষ্ট্য বলে দিতে পারি। প্রথমে আমরা নিচের কোডটি টাইপ করে রান করবো, তারপর সেটি আলোচনা করবো।

class Car:

name = ""

color = ""

def \_\_init\_\_(self, name, color):

self.name = name

self.color = color

def start(self):

print("Starting the engine")

my\_car = Car("Corolla", "White")

print(my\_car.name)

print(my\_car.color)

my\_car.start()

প্রোগ্রামটি রান করলে ঠিকঠাক আউটপুট আসবে। এখানে আমি একটি নতুন মেথড ব্যবহার করেছি, \_\_init\_\_()। যখন কোনো ক্লাসের অবজেক্ট তৈরি করা হয়, তখন এই মেথডটি আপনাআপনি (অটোমেটিক) কল হয়। তাই যখন my\_car = Car(“Corolla”, “White”) স্টেটমেন্টটি চলে, তখন আসলে Car ক্লাসের \_\_init\_\_ মেথড কল হয়। সেই মেথডের প্রথম প্যারামিটার হচ্ছে self। এটি সবসময়ই দিতে হবে। তারপরে যেহেতু আমরা গাড়ির নাম ও রং সেট করে দিতে চাই, তাই name ও color নামে দুটি প্যারামিটার রাখছি। তারপরে ফাংশনের ভেতরে self.name-এ name অ্যাসাইন করছি, আর self.color-এ color অ্যাসাইন করছি। আমি যখন লিখছি self.name, সেটি বোঝাচ্ছে যে, যেই অবজেক্টটি আমি তৈরি করছি, তার name অ্যাট্রিবিউট। এখানে Car ক্লাসের ভেতরে যে name ডিক্লেয়ার করেছি (name = “”), self.name, আর init ফাংশনের name প্যারামিটার – তিনটি কিন্তু আলাদা। এখানে এসে ব্যাপারটা একটু এলোমেলো লাগতে পারে। তাই আমরা ওপরের কোডটি একটু পরিবর্তন করে লিখি।

class Car:

def \_\_init\_\_(self, n, c):

self.name = n

self.color = c

def start(self):

print("Starting the engine")

my\_car = Car("Corolla", "White")

print(my\_car.name)

print(my\_car.color)

my\_car.start()

প্রোগ্রামটি রান করলে আউটপুট আসবে এরকম –

$ python car.py

Corolla

White

Starting the engine

এখানে যখন আমি my\_car = Car(“Corolla”, “White”) লিখছি, তখন \_\_init\_\_(self, n, c)-এর self-এ যাচ্ছে my\_car অবজেক্টের রেফারেন্স, n-এ যাচ্ছে “Corolla”, আর c-তে যাচ্ছে “White”। তারপর যখন self.name = n স্টেটমেন্ট এক্সিকিউট হচ্ছে, তখন my\_car অবজেক্টের name নামে একটি অ্যাট্রিবিউট তৈরি হচ্ছে আর সেখানে n অ্যাসাইন হচ্ছে। একইভাবে color অ্যাট্রিবিউট তৈরি হয়ে সেখানে c-তে যা পাঠিয়েছিলাম, তা অ্যাসাইন হচ্ছে।

Car ক্লাসে যে দুটি ডেটা অ্যাট্রিবিউট তৈরি করেছিলাম (name ও color), সেগুলো কিন্তু আমি বাদ দিয়ে দিয়েছি। কারণ আমি অবজেক্ট তৈরি করার সময় init মেথডের ভেতরে self.name ও self.color যখন লিখছি, তখন সেই অবজেক্টের জন্য name ও color নামে অ্যাট্রিবিউট তৈরি হয়ে যাচ্ছে। একে বলে ইনস্ট্যান্স অ্যাট্রিবিউট, যা কেবল ওই ক্লাসের ইনস্ট্যান্সের (বা অবজেক্টের) থাকে। তবে এখন কিন্তু আর ক্লাসের নাম লিখে ডট দিয়ে name (ও color) একসেস করা যাবে না। আমরা যদি লিখি print(Car.name), তাহলে আমরা এরর পাবো : AttributeError: type object ‘Car’ has no attribute ‘name’। অর্থাৎ Car ক্লাসের name নামে কোনো অ্যাট্রিবিউট নেই। আমরা যখন এরকম লিখেছিলাম –

class Car:

name = ""

color = ""

তখন, এই name ও color–কে বলা হয় ক্লাস অ্যাট্রিবিউট। বেশিরভাগ সময়ই আমাদের এরকম অ্যাট্রিবিউট তৈরি করার দরকার পরবে না।

এখন ক্লাসের ভেতরে যেসব মেথড আছে, তারা যদি ওই ক্লাসের অবজেক্টের বিভিন্ন অ্যাট্রিবিউট একসেস করতে চায়, সেই কাজটি করতে পারবে, তবে অ্যাট্রিবিউটের নামের আগে অবশ্যই self লিখে ডট দিতে হবে, তাহলে পাইথন বুঝতে পারবে যে, অ্যাট্রিবিউটটি কোন অবজেক্টের সঙ্গে সংশ্লিষ্ট।

class Car:

def \_\_init\_\_(self, n, c):

self.name = n

self.color = c

def start(self):

print("name: ", self.name)

print("color: ", self.color)

print("Starting the engine")

my\_car = Car("Corolla", "White")

my\_car.start()

ওপরের প্রোগ্রামে আমরা start() মেথডের ভেতরে অবজেক্টের নাম ও রং প্রিন্ট করেছি। এখন একটি মজার জিনিস দেখাই। self-এ যে অবজেক্ট পাঠানো হয়, তার প্রমাণ পাওয়া যাবে নিচের প্রোগ্রাম চালালে –

class Car:

def \_\_init\_\_(self, n, c):

self.name = n

self.color = c

def start(self):

print("name: ", self.name)

print("color: ", self.color)

print("Starting the engine")

my\_car = Car("Corolla", "White")

Car.start(my\_car)

আমরা my\_car অবজেক্ট তৈরি করলাম। তারপর, Car.start() মেথড কল করলাম কিন্তু আর্গুমেন্ট হিসেবে my\_car অবজেক্ট ব্যবহার করলাম। এই প্রোগ্রামের আউটপুট কিন্তু হুবুহু আগের প্রোগ্রামের মতোই হবে। তবে বলে রাখা ভালো যে, সাধারণত এভাবে আমরা start() মেথড কল করবো না।

এখন আমরা চাইলে একাধিক কার অবজেক্ট তৈরি করতে পারি।

class Car:

def \_\_init\_\_(self, n, c):

self.name = n

self.color = c

def start(self):

print("name: ", self.name)

print("color: ", self.color)

print("Starting the engine")

my\_car1 = Car("Corolla", "White")

my\_car1.start()

my\_car2 = Car("Premio", "Black")

my\_car2.start()

my\_car3 = Car("Allion", "Blue")

my\_car3.start()

প্রোগ্রামটি রান করলে আউটপুট আসবে এরকম :

$ python car.py

name: Corolla

color: White

Starting the engine

name: Premio

color: Black

Starting the engine

name: Allion

color: Blue

Starting the engine

প্রতিটি অবজেক্টের অ্যাট্রিবিউটগুলো কিন্তু আলাদা। যেমন, my\_car1-এর color আর my\_car2-এর color আলাদা। এটি হচ্ছে অবজেক্ট ওরিয়েন্টেড প্রোগ্রামিংয়ের একটি বড় সুবিধা। আমরা যত খুশি অবজেক্ট তৈরি করতে পারি। প্রতিটি অবজেক্ট তার নিজস্ব অ্যাট্রিবিউট ধারণ করবে।

আরেকটি কথা বলা প্রয়োজন। আমরা যে \_\_init\_\_ মেথড ব্যবহার করছি, যা অবজেক্ট তৈরি হওয়ার সময় আপনাআপনি কল হয়, একে বলে কনস্ট্রাকটর (constructor)।

এখন ধরা যাক, Car ক্লাসের অবজেক্ট তৈরি করার পর আমার খেয়াল হলো, অবজেক্টের একটি ডেটা অ্যাট্রিবিউট দরকার, যেটি ক্লাসের মধ্যে নেই। আর আমি এখন ক্লাসটিও পরিবর্তন করতে চাইছি না। কিন্তু তাতে কোনো সমস্যা নেই। আমি চাইলে যেকোনো সময় আমাদের অবজেক্টের সঙ্গে একটি ডেটা অ্যাট্রিবিউট জুড়ে দিতে পারবো।

class Car:

def \_\_init\_\_(self, n, c):

self.name = n

self.color = c

def start(self):

print("Starting the engine")

car = Car("Corolla", "White")

car.year = 2017

print(car.name, car.color, car.year)

**ক্লাস ইনহেরিটেন্স**

ইনহেরিটেন্স (inheritance) যেকোন অবজেক্ট ওরিয়েন্টেড প্রোগ্রামিং এর জন্য এক অনন্য ফিচার (feature).

এটি খুবই একটি সাধারণ ধারণা। ধরা যাক, আপনার বাবার একটি গাড়ি আছে, অর্থাৎ এটি আপনারও গাড়ি, আবার আপনার নিজের কেনা একটি গাড়িও থাকতে পারে। এখানে আপনার বাবার গাড়িটি আপনার কাছে ইনহেরিট হচ্ছে।

প্রোগ্রামিং এর ক্ষেত্রেও তাই। যখন একটি প্যারেন্ট ক্লাসের (Parent Class) কোন একটি অ্যাট্রিবিউট (Attribute) বা মেথড (Method), চাইল্ড ক্লাসে (Child Class) ইনহেরিট হয় তখন তাকে**ইনহেরিটেন্স বলে**।

আগের পর্বে আমরা শুধুমাত্র একটা ক্লাস নিয়ে আলোচনা করেছি। এই পর্বে আমরা একাধিক ক্লাস ব্যবহার করে ইনহেরিটেন্স এর উদাহরণ দেখে নেব।

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | # This is an example of inheritance in Python  class MathClass: # Parent Class      def \_\_init\_\_(self,x,y):          self.x=x          self.y=y      def Multiply\_function(self):          z=self.x\*self.y          return z    class MathClass2(MathClass): # Child Class      def \_\_init\_\_(self,x,y):          self.x=x          self.y=y      def Division\_function(self):          z=self.x/self.y          return z    FirstObject=MathClass(5,10) # creating an object  SecondObject=MathClass2(4,2) # creating an object    print(FirstObject.Multiply\_function()) # Accessing a method  print(SecondObject.Division\_function()) # Accessing a method  print(SecondObject.Multiply\_function()) # Accessing a method using inheritance |

ফলাফল-

50

2.0

8

এখানে দুইটা ক্লাস আছে, MathClass এবং MathClass2. এখন ২৩ নম্বর লাইনে SecondObject এ Multiply\_function() কল করা হয়েছে; কিন্তু MathClass2 তে এই নামে কোন ফাংশন নেই। মূলত এই ফাংশনটা MathClass থেকে ইনহেরিট হয়েছে।

আগে আমরা self কিওয়ার্ডটি ব্যবহার করছিলাম, কিন্ত ইনহেরিটেন্স এর জন্য ক্লাসের ভিতর যে ক্লাসকে ইনহেরিট করতে চাচ্ছি সেটার নাম দিতে হয় যেটা ১০ নম্বর লাইনে দেখানো হয়েছে।

প্যারেন্ট ক্লাস (Parent Class)

যে ক্লাসকে ইনহেরিট করা হবে সেটাকে প্যারেন্ট ক্লাস বলা হয়। উপরের উদাহরণে ২য় লাইনে যে MathClass ব্যবহার করা হয়েছে এটা প্যারেন্ট ক্লাস বা বেজ ক্লাস (Base Class)।

চাইল্ড ক্লাস (Child Class)

যে ক্লাসে কোন ক্লাসকে ইনহেরিট করা হবে তাকে চাইল্ড ক্লাস বলা হয়। উপরের উদাহরণের ১০ম লাইনে MathClass2 একটি চাইল্ড ক্লাস।

**রিসার্ভড কীওয়ার্ড কিভাবে ব্যবহার করবো ?**

এই অধ্যায়ে আপনি পাইথন কীওয়ার্ড(keywords) সম্মন্ধে জানবেন। কীওয়ার্ড হলো পাইথনের সংরক্ষিত শব্দ(word) যা সিনট্যাক্স এর অংশ।

আপনি [ভ্যারিয়েবল](https://www.sattacademy.com/python/python-variables.php) এবং [ফাংশন](https://www.sattacademy.com/python/python_functions.php) এর নাম দেওয়ার জন্য কীওয়ার্ড অথবা [আইডেন্টিফায়ার](https://www.sattacademy.com/python/python-identifier.php) ব্যবহার করতে পারবেন না। এগুলো পাইথন ভাষার সিনট্যাক্স এবং গঠন নির্ধারণের জন্য ব্যবহৃত হয়।

পাইথন এর কীওয়ার্ডসমূহ কেস-সেনসিটিভ(case-sensitive)।

পাইথন ৩.৬ এ ৩৩ টি কীওয়ার্ড আছে। এগুলো ভার্সন পরিবর্তনের সময় খুব সামান্যই পরিবর্তন হয়।

True, False এবং None কীওয়ার্ড ছাড়া অন্য সকল কিওয়ার্ডসমূহ ছোট হাতের অক্ষর(lowercase) লিখতে হয় এবং লিখার সময় হুবুহু লিখতে হবে। নিম্নে কীওয়ার্ডসমূহের তালিকা দেওয়া হলো।

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| পাইথন ভাষার কীওয়ার্ডসমূহ | | | | |
| False | class | finally | is | return |
| None | continue | for | lambda | try |
| True | def | from | nonlocal | while |
| and | del | global | not | with |
| as | elif | if | or | yield |
| assert | else | import | pass |  |
| break | except | in | raise |  |

#### ফাইল পরিচিতি

# **পাইথন ফাইল রিডিং এবং রাইটিং**

আমরা যখন কোন প্রোগ্রাম লিখি, রান করি সাধারণত ডেটা গুলো স্ক্রিনে দেখায়। এক সময় হারিয়ে যায়। আমরা চাই আমাদের প্রোগ্রাম যে ডেটা ইউজার থেকে ইনপুট নিবে, তা সেভ করে রাখতে, যেন ভবিষ্যৎ এও ব্যবহার করা যায়। আবার মাঝে মাঝে কোন ফাইল থেকে ডেটা পড়ে কোন অপারেশন করার দরকার হতে পারে। আর তার জন্য পাইথনে রয়েছে ফাইল রিডিং এবং রাইটিং অপশন।

কোন ফাইল থেকে ডেটা পড়া বা তাতে কিছু ডেটা রাখার জন্য আগে তা ওপেন করতে হয়। তা করা হয় open() দিয়ে। আর open() একটা অবজেক্ট রিটার্ণ করে। ওপেন এর ভেতর ফাইল এর নাম দিতে হয়। ধরে নিচ্ছি আমরা যে ডিরেক্টরিতে পাইথন প্রোগ্রামটি রেখেছি, ঐ একই ডিরেক্টরি/ফোল্ডারে file.txt নামক একটা ফাইল রয়েছে। যদি না থাকে, তাহলে একটা txt ফাইলে তৈরি করে নিন। file.txt এর ভেতরে কিছু লেখা রাখুন।

|  |  |
| --- | --- |
| 1  2  3 | f = open('file.txt')  content = f.read()  print(content ) |

এখানে প্রথম লাইনে আমরা ফাইলটি অবজেক্ট হিসেবে ওপেন করেছি।এরপর তার ভেতরের কন্টেন্ট গুলো রিড করেছি। এবং তা content নামক একটা ভ্যারিয়েবলে রেখেছি। শেষে কন্টেন্ট ভ্যারিয়েবলটি প্রিন্ট করেছি।

ফাইলের ভেতরে যাই থাকবে, তাই আমাদের আউটপূট দিবে।

এখন যদি আমরা ফাইলের ভেতরের শুধু প্রথম লাইন পড়তে চাই, তার জন্যঃ

|  |  |
| --- | --- |
| 1  2  3 | f = open('file.txt')  line = f.readline()  print(line) |

সব গুলো লাইন একটা একটা করে রিড করার জন্য আমরা একটা লুপ চালাতে পারি, যেমনঃ

|  |  |
| --- | --- |
| 1  2  3 | f = open( 'file.txt')  for line in f:      print(line) |

এখানে আমরা for লুপ ব্যবহার করেছি। for লুপ সম্পর্কে কন্ট্রোল স্টেটম্যান্ট চ্যাপ্টার থেকে জানা যাবে।

**ফাইলে কিছু ডেটা রাখা / ফাইল রাইটিংঃ**

আমরা যখন open() ব্যবহার করে কোন ফাইল ওপেন করি, তা ডিফল্ট ভাবে রিডিং মুডে ওপেন হয়। যেমন f = open( ‘file.txt’) এখানে আমাদের file.txt ফাইলটি রিডিং মুডে ওপেন হয়েছে। আমরা শুধু ফাইল থেকে ডেটা পড়তে পারব। কিন্তু কোন কিছু ফাইলে রাখতে পারব না। যদি আমরা কোন কিছু লিখতে চাই ফাইলে, তার জন্য মুড পরিবর্তন করতে হবে। যেমন রাইটিং মুডে ওপেন করার জন্যঃ

|  |  |
| --- | --- |
| 1 | f = open( 'file.txt','w') |

এখন ফাইলটি রাইটিং মুডে ওপেন হয়েছে। এবার আমরা এর ভেতরে কিছু লিখবঃ

|  |  |
| --- | --- |
| 1  2 | f = open( 'file.txt','w')  f.write('This is a testn') |

ফাইল নিয়ে কাজ করা শেষ হলে আমরা ফাইলটি ক্লোজ করব। close() ব্যবহার করে। এখন থেকে আমরা ফাইল ওপেন করে কাজ করা শেষে close() কল করে ফাইলটি ক্লোজ করব।

এখন যদি আমরা file.txt ফাইলটি ওপেন করে দেখি, তাহলে তার ভেতরে দেখতে পাবোঃ

|  |  |
| --- | --- |
| 1 | This is a test |

আগের ডেটা গুলো রিমুভ হয়ে গিয়েছে।

আমরা যদি ফাইলে কোন নাম্বার রাখতে চাই, তাহলে নাম্বারকে আগে স্ট্রিং এ কনভার্ট করে নিতে হবে, তারপর তা ফাইলে রাখতে পারব। যেমন আমরা pi এর মান ফাইলে রাখতে চাই, তার জন্যঃ

|  |  |
| --- | --- |
| 1  2  3  4  5 | pi = 3.14159265359  s = str(pi)  f = open( 'file.txt', 'w' )  f.write(s)  f.close() |

ফাইলটি ওপেন করলে এখন দেখতে পাবো পাই এর মান স্টোর করা হয়েছে। আমরা চাইলে পাই এর মানের সাথে কোন স্ট্রিং ও যোগ করে রাখতে পারিঃ

|  |  |
| --- | --- |
| 1  2  3  4  5 | pi = 3.14159265359  s = str(pi)  f = open( 'file.txt', 'w' )  f.write( 'Value of pi: ' + s)  f.close() |

এটা ফাইলে স্টোর কবেঃ Value of pi: 3.14159265359

এগুলো খুব সাধারণ কাজ। যদিও এই সাধারণ অপারেশন ব্যবহার করে আমরা অনেক গুলো কাজ করে ফেলতে পারি।

**ফাইল নিয়ে আরো কিছু কাজ**

ফাইল নিয়ে বলার মত তো কিছুই নাই। সরকারি দফতরে ফাইল নিয়ে যে কত্ত ধকল পোহাইতে হইছে তা কেবল আল্লাহ মালুম। এক টেবিল থেকে অন্য টেবিলে তো দূরে থাক, একই টেবিলে ফাইলের স্তুপে নিচ থেকে উপরে উঠাইতেও কি আর কম ধকল? তবে পাইথনে ফাইল নিয়ে কাজবাজ করা এতটা কঠিন না। ফাইল নিয়ে কাজ করতে গিয়ে আমরা তিনটা জিনিস শিখব। ফাইল ওপেন (open) করা, ওপেন করা ফাইল রিড (read) করা, রিড করার পর ফাইল রাইট (write) করা আর শেষমেষ ফাইল ক্লোজ (close) করা।

প্রথমে আমরা শিখব কিভাবে সাধের ফাইলটা ওপেন করা যায়। কারণ, ফাইল ওপেন না করে তো আর বাকি কাজ করা যাবে না। পাইথনে ফাইল ওপেন করার জন্য open() ফাংশনটা ব্যবহার করব আমরা। এই ফাংশনটা তিনটা প্যারামিটার নিয়ে কাজ করে।

* ফাইলের নাম: প্রথম প্যারামিটার হল ফাইলের নাম। আমাদের স্ক্রিপ্ট ও আলোচ্য ফাইল যদি একই ডিরেক্টরিতে হয় তবে শুধু ফাইলের নামটা স্ট্রিং হিসাবে দিতে হয়। ফাইল যদি আলাদা ডিরেক্টরিতে থাকে তবে পুরো পাথটা স্ট্রিং হিসাবে দিতে হবে।
* অ্যাক্সেস মোড: দ্বিতীয় প্যারামিটার হল অ্যাক্সেস মোড। একটা ফাইলকে আমরা বিভিন্ন কাজের জন্য ওপেন করতে পারি। সেক্ষেত্রে ভিন্ন ভিন্ন কাজের জন্য ভিন্ন ভিন্ন মোডে ফাইলটা ওপেন করতে হবে। নিচে কতগুলো মোডের নাম ও কোন কাজে লাগে তা দেয়া হল।

| মোড | বর্ণনা |
| --- | --- |
| r | এটা হল ডিফল্ট মোড। কোন মোড স্পেসিফাই করে না দেয়া হলে এটা আপনা-আপনি কাজ করে। এটা শুধু রিড করার জন্য একটা ফাইলকে ওপেন করে। আর ফাইল পয়েন্টার ফাইলের শুরুতে থাকে। |
| rb | সবকিছু মোড r এর মতই। শুধু তফাৎ হল ফাইলটা বাইনারি ফরম্যাটে ওপেন করে। |
| r+ | সবকিছু মোড r এর মতই। এক্সট্রা বেনেফিট হল ফাইলটা রাইটও করা যায়। |
| rb+ | r+ আর rb এর কম্বিনেশন। |
| w | ফাইলকে রাইট মোডে ওপেন করে। ফাইলের অস্তিত্ব না থাকলে নিজ থেকেই ফাইল তৈরি করে নেয়। আর অস্তিত্ব থেকে থাকলে পুরাতন ফাইলকে ওভাররাইট করে। |
| wb | সবকিছু মোড w এর মতই। শুধু তফাৎ হল ফাইলটা বাইনারি ফরম্যাটে ওপেন করে। |
| w+ | সবকিছু মোড w এর মতই। এক্সট্রা বেনেফিট হল ফাইলটা রিডও করা যায়। |
| wb+ | w+ আর wb এর কম্বিনেশন। |
| a | ফাইলকে অ্যাপেন্ড (append) মোডে ওপেন করে। এই মোডে ফাইল পয়েন্টার ফাইলের একেবারে শেষে থাকে। ফাইল রাইট করলে তা ফাইলের শেষে যোগ হয়। ফলে আগের সব ডাটা রক্ষিত থাকে। তবে ফাইলের অস্তিত্ব না থাকলে নিজ থেকেই ফাইল তৈরি করে নেয়। |
| ab | সবকিছু মোড a এর মতই। শুধু তফাৎ হল ফাইলটা বাইনারি ফরম্যাটে ওপেন করে। |
| a+ | সবকিছু মোড a এর মতই। এক্সট্রা বেনেফিট হল ফাইলটা রিডও করা যায়। |
| ab+ | a+ আর ab এর কম্বিনেশন। |

* বাফারিং: তৃতীয় প্যারামিটার হল বাফারিং। এটা একটু উচ্চতর বিষয়। আমরা এটা সম্পর্কে পরে জানব।

কয়েকটা উদাহরণ দিলে বিষয়গুলো আরো পরিষ্কার হয়ে যাবে আমাদের কাছে। এজন্য আমরা ছোট্ট একটা কাজ করব। আমাদের স্ক্রিপ্টের সাথে একই ডিরেক্টরিতে test.txt নামে একটা ফাইল তৈরি করব। তারপর ফাইলে The name of my country is Bangladesh. কথাটা লিখে সেইভ করে রাখব। এবার কয়েকটা প্রোগ্রাম লেখা যাক।

my\_file = open('test.txt', 'r')

content = my\_file.read()

print(content)

my\_file.close()

**আউটপুট**

The name of my country is Bangladesh.

প্রোগ্রামটার ময়নাতদন্ত করা যাক একটু। প্রথম লাইনে আমরা ফাইলটা রিড মোডে ওপেন করেছি। দ্বিতীয় লাইনে ফাইলটা read() ফাংশনের (আসলে মেথড) সাহায্যে রিড করে ডাটা content ভ্যারিয়েবলে অ্যাসাইন করেছি। আর শেষ লাইনে close() ফাংশন (মেথড) ব্যবহার করে ফাইলটা ক্লোজ করে দিয়েছি। আরেকটা প্রোগ্রাম দেখা যাক।

my\_file = open('test.txt', 'w')

my\_file.write('I am Maksudur Rahman Maateen.')

my\_file.close()

এখানে write() ফাংশন (মেথড) ব্যবহার করে আমরা ফাইলটা রাইট করেছি। এই ফাংশনের ভিতর স্ট্রিং হিসাবে যা দেয়া হবে তাই রাইট হবে ফাইলে। প্রোগ্রামটা রান করলে আমরা আউটপুট কিছুই পাব না। তবে ফাইলটা যদি নরমালি ওপেন করে দেখি তাহলে দেখব সেখানে এখন The name of my country is Bangladesh. এর পরিবর্তে I am Maksudur Rahman Maateen. লেখা রয়েছে। এবার আরেকটা প্রোগ্রাম দেখা যাক।

my\_file = open('test.txt', 'a')

my\_file.write('I am from Bangladesh.')

my\_file.close()

এই প্রোগ্রামটা রান করলেও আমরা আউটপুট কিছুই পাব না। তবে ফাইলটা যদি নরমালি ওপেন করে দেখি তাহলে দেখব সেখানে এখন I am Maksudur Rahman Maateen. এর পরে I am from Bangladesh. লেখা রয়েছে। এবার আরেকটা প্রোগ্রাম দেখা যাক।

my\_file = open('test.txt', 'r')

content = my\_file.read(5)

print(content)

content = my\_file.read()

print(content)

position = my\_file.tell()

print(position)

my\_file.seek(0, 0)

content = my\_file.read()

print(content)

my\_file.close()

**আউটপুট**

I am

Maksudur Rahman Maateen.I am from Bangladesh.

51

I am Maksudur Rahman Maateen.I am from Bangladesh.

লম্বা-চওড়া একটা প্রোগ্রাম। আমরা এবার এটা ব্যাখায় যাব। দ্বিতীয় লাইনে read() এর ভিতর আমরা 5 ভ্যালু পাস করেছি। এর ফলে ফাইলের প্রথম ৫ টা ক্যারেক্টার পর্যন্ত রিড হবে শুধু। একটা কথাটা তো আমরা নিশ্চয়ই জানি, স্পেসও একটা ক্যারেক্টার। যাহোক, ষষ্ঠ লাইনে এসে আমরা tell() ফাংশন দিয়ে ফাইল পয়েন্টারের পজিশন খুঁজে বের করেছি। আর অষ্টম লাইনে seek() ফাংশন দিয়ে ফাইল পয়েন্টার আবার শুরুতে নিয়ে গিয়েছি। আরেকটা উদাহরণ দেখা যাক।

with open('test.txt', 'r') as my\_file:

content = my\_file.read()

print(content)

**আউটপুট**

I am Maksudur Rahman Maateen.I am from Bangladesh.

এই প্রোগ্রামে আমরা with স্টেটমেন্ট ব্যবহার করেছি। স্টেটমেন্টটা ব্যবহার করে ফাইলটা my\_file নামে ওপেন করেছি। বাদবাকি অপারেশন নরমাল। কেউ কি লক্ষ্য করেছি যে আমরা ফাইলটা ক্লোজ করিনি। আসলে এটাই এই স্টেটমেন্টের সুবিধা। কাজ শেষে বা কোন এরর পেলে নিজ থেকেই ফাইলটা ক্লোজ করে দেয়।

এই ছিল ফাইল নিয়ে আমাদের বেসিক আলোচনা। এখন বাড়ির কাজ আছে। একটা ফাইলে লুপ ঘুরিয়ে ১ থেকে ১০০ পর্যন্ত লাইন বাই লাইন রাইট করব আমরা। কি পারব তো সবাই? চেষ্টা করলে অবশ্যই পারব।

**লিস্ট কম্প্রিহেনশন**

লিস্ট শব্দের বাংলা অর্থ তালিকা। আমাদের বোধহয় ব্যখ্যা করার দরকার পড়ে না তালিকা কি জিনিস । পাইথনেও লিস্ট একই কাজ করে । সহজ কথায় লিস্ট হল কতগুলো আইটেমের একটি তালিকা । অনেক প্রোগ্রামিং ল্যাঙ্গুয়েজে লিস্ট ডিক্লেয়ার করার সময় বলে দিতে হয় লিস্টের আইটেমগুলোর টাইপ কি হবে, পাইথনে তার দরকার পড়ে না । একটি লিস্টের আইটেমগুলো বিভিন্ন টাইপের হতে পারে ।

কিভাবে লিস্ট ডিক্লেয়ার করব? (থার্ড) ব্রাকেটের ভিতরে কমা দিয়ে একেকটি আইটেম সেপারেট করে দিলেই লিস্ট তৈরি হয়ে যাবে । আসুন উদাহরণ দেখি:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | my\_list = [1,"a string",45.56]  print my\_list[0]  print my\_list[1]  print my\_list[2]    print my\_list    print type(my\_list[0])  print type(my\_list[1])  print type(my\_list[2]) |

প্রথমে কোডগুলো মনোযোগ দিয়ে পড়ুন । বোঝার চেষ্টা করূন এর আউটপুট কি হতে পারে ।

বরাবরের মত একটি পাইথন ফাইলে এই কোডগুলো লিখে রান করে দেখুন কি আউটপুট দেখায় । type() ফাংশনটির ব্যবহার আমরা আগেই দেখেছি । আউটপুট দেখে মিলিয়ে নিন আপনি কি আশা করেছিলেন আউটপুট হিসেবে আর কি এসেছে আউটপুট । যদি না মিলে, বোঝার চেষ্টা করূন কোথায় বুঝতে পারেন নি ।

এই কোড থেকে আমরা কি কি দেখলাম:

* কিভাবে লিস্ট ডিক্লেয়ার করতে হয়
* লিস্টের আইটেমগুলোর একটি ইন্ডেক্স ভ্যালু থাকে। এই ইন্ডেক্স ভ্যালু ব্যবহার করে আমরা n-তম আইটেমের মান বের করতে পারি
* এই ভ্যালুর মান 0 থেকে শুরু হয় । অর্থাৎ প্রথম আইটেমের ইন্ডেক্স 0, দ্বিতীয়টির 1 এভাবে n-তম আইটেমের ইন্ডেক্স (n-1)

লিস্ট সম্পর্কে আরো জানার আগে আমরা range() ফাংশনটির ব্যবহার দেখে নেই । এই ফাংশনটির একটি উদাহরণ দেখি :

|  |  |
| --- | --- |
| 1  2 | print range(0,10)  print range(0, 100, 10) |

এই ফাংশনটি সংখ্যার লিস্ট তৈরি করে । এর সিগনেচার অনেকটা এরকম: range(min,max,step) । এখানে min হল নূন্যতম ভ্যালু যেটা থেকে লিস্ট শুরু হবে । max হল সর্বোচ্চ ভ্যালু যেটার ঠিক আগের ভ্যালু পর্যন্ত লিস্ট তৈরি হবে । step হল মধ্যবর্তী ব্যবধান ।

উপরোক্ত কোড রান করালে প্রথমে আমরা পাব 0 থেকে শুরু করে 10 এর ঠিক আগের ভ্যালু অর্থাৎ 9 পর্যন্ত । যদি step না দেওয়া হয় তাহলে পাইথন এটার ভ্যালু 1 ধরে নেয় । দ্বিতীয় বার আমরা step হিসেবে 10 দিয়েছি । তাই এবার আমরা 0 থেকে শুরু করে প্রতি 10 ঘর পর পর সংখ্যার লিস্ট পাব 90 পর্যন্ত ।

আমরা লিস্ট প্র্যাকটিস করার জন্য range() ফাংশনটি ব্যবহার করে দ্রুত লিস্ট তৈরি করে নিব । আসুন ফেরা যাক লিস্টে । আমরা দেখেছি কিভাবে ইন্ডেক্স ব্যবহার করে আমরা লিস্টের আইটেমগুলো এক্সেস করেছি । ধরূন আমাদের লিস্টের সব ডাটা লাগবে না, আমরা একটি নির্দিষ্ট রেঞ্জ নিয়ে কাজ করতে চাই । পাইথন আমাদের সেই সুবিধা দেয় (যা অন্য অনেক ল্যাঙ্গুয়েজে পাওয়া যায় না ) । আসুন দেখি কিভাবে:

এই উদাহরণটি নিজেরা চেষ্টা করার জন্য প্রথমেই একটি লিস্ট তৈরি করে নেই ।

|  |  |
| --- | --- |
| 1 | sl = range(1,11) # A list containing the integers from 1 to 10 |

আসুন এবার লিস্ট নিয়ে নাড়া চাড়া করা যাক:

|  |  |
| --- | --- |
| 1  2  3  4  5 | list1to5 = sl[0:5]  print list1to5    list2to7 = sl[1:7]  print list2to7 |

এই কোড রান করালে দেখা যাবে list1to5 একটি লিস্ট যার ভ্যালু 1 থেকে 5 । sl[0:5] বলতে বোঝানো হয় sl নামক লিস্টের 0-তম আইটেম থেকে শুরু করে 5-তম আইটেমের আগের আইটেম পর্যন্ত আইটেমগুলো নিয়ে তৈরি একটি লিস্ট । এবার নিজে নিজেই বোঝার চেষ্টা করুন list2to7 এর ভ্যালু কি হতে পারে এবং কেন ।

এবার নিজে কিছু কাজ করুন:  
3 থেকে 9 পর্যন্ত লিস্ট বোঝাতে আমরা কি লিখব?  
sl[:5] এর ভ্যালু কত হবে?  
sl[4:] এর ভ্যালু কত হবে?  
sl[:] এর ভ্যালু কত হবে? কেন?

আমরা range ফাংশনে step এর ব্যবহার দেখেছিলাম । লিস্টের ক্ষেত্রেও step ব্যবহার করা যায় । যেমন:

|  |  |
| --- | --- |
| 1  2 | print sl[0:10:2]  print sl[0:9:3] |

অর্থাৎ শেষে আরেকটি কোলন দিয়ে আমরা step ভ্যালুটি নির্দেশ করে থাকি । তাই প্রথম ক্ষেত্রে আমরা 0-তম আইটেম থেকে শুরু করে 2টি আইটেম বাদ দিয়ে দিয়ে 10-তম আইটেমের আগের আইটেম পর্যন্ত যে আইটেগুলো আছে সেগুলোর লিস্ট পাব । নিজে নিজে বোঝার চেষ্টা করি দ্বিতীয় ক্ষেত্রে কি ঘটছে ।

যে কোন ভ্যালুর আগে মাইনাস চিহ্ন দিলে তার অবস্থান বিপরীত দিক থেকে বিবেচনা করা হয় । তাই শেষ দিক থেকে 5-তম আইটেমের ভ্যালু হবে sl[-5] । এভাবে শেষ দিক থেকে 2-তম আইটেমের আগ পর্যন্ত আইটেমগুলোর লিস্ট পাব: sl[:-2] । step এর ভ্যালু নেগেটিভ হলে গনণা উল্টো দিকে হবে । যেমন শেষ দিক থেকে 2-তম আইটেমের আগের আইটেম থেকে শুরু করে 3-তম আইটেম পর্যন্ত আইটেমগুলো 2 ধাপ করে পেছালে আমরা যে লিস্টটি পাব তার জন্য আমাদের কে লিখতে হবে : sl[-2:3:-2]

এভাবে নিজেরা ইচ্ছামত লিস্ট তৈরি করে তার বিভিন্ন অংশ আলাদা করার চেষ্টা করি । প্রথমবার দেখে লিস্টের সিন্ট্যাক্স খুব জটিল মনে হতে পারে । কিন্তু কিছুদিন অনুশীলন করলেই ঠিক হয়ে যাবে । পাইথনের চমৎকার ফিচারগুলোর মধ্যে অন্যতম হল লিস্ট এর এই ব্যবহার । একটি লিস্ট এর যে কোন অংশ নিয়ে আরেকটি লিস্ট খুব সহজেই তৈরি করা যায়। পাইথনে লিস্টের আরো চমকপ্রদ কিছু ব্যবহার রয়েছে যেগুলো নিয়ে ভবিষ্যতে কোন এক সময় লিখব ।

#### ডিক্শনারি কম্প্রিহেনশন

ডিকশনারি (Dictionary) পাইথনের একটি অন্যতম built-in  ডেটা টাইপ।  লিস্ট (list ) এর মত  এটিও একটা গুরুত্বপূর্ণ ডেটা টাইপ। আসলে ডিকশনারি ( **key:value pair**) বা **key: value জোড়া**ছাড়া আর কিছুই না । পাইথনের সিকোয়েন্স টাইপ যেমনঃ লিস্ট (list) কে আমরা নাম্বার (number) দিয়ে  ইনডেক্স (index) করে  থাকি  কিন্তু ডিকশনারি (Dictionary ) কে ইনডেক্স করা হয় key ব্যবহার করে ।  আর এখানে একজন প্রোগ্রামারকে  অবশ্যই  মনে রাখতে হবে keys গুলোকে অবশ্যই  immutable  টাইপের হতে হবে।  আর এটিই সবচেয়ে গুরুত্বপূর্ণ। immutableবলতে বোঝায় যেগুলোকে পরিবর্তন করা যায় না। মূলত পাইথনের ***immutable***টাইপ গুলোর মধ্যে String , Number কে  সাধারণত ***key***হিসেবে ব্যবহার করা হয়। নিচে পাইথনের ***immutable***আর ***immutable***টাইপ গুলো দেওয়া হল  আর এগুলোকে অবশ্যই  মাথায় রাখতে হবে।

#### ডেকরেটর সম্পর্কে প্রাথমিক ধারণা

“ডেকরেটর! ওই যে বিভিন্ন অনুষ্ঠানের সাজ সজ্জা করে মানে ডেকরেট করে….. তো পাইথনে আবার তাদের কি কাজ!”

পোস্টের টাইটেল দেখে যদি আপনার মনে এরকম প্রশ্নের উদয় হয় তাহলেও দোষের কিছু নেই। পাইথনে ডেকরেটর কিছুটা এডভান্স আর কমপ্লেক্স টপিক। তবে চিন্তা নেই, এই পোস্টে আমরা একটু সহজ ভাবে, ধাপে ধাপে জানার চেষ্টা করবো পাইথনে ডেকরেটর জিনিষটা কী, কীভাবে কাজ করে আর কীভাবেই বা ব্যাবহার করে।

@mydecorator

def myfunc():

      pass

ঠিক ঠাক ভাবে বললে, ডেকরেটর হল এক ধরনের callable যা অন্য callable এর ফাংশনালিটিকে মডিফাই করে। আরেকটু সহজ করে বললে, ডেকরেটর হল এক ধরনের ফাংশন যা অন্য ফাংশনের ফাংশনালিটিকে মডিফাই করে। [ডেকরেটর ক্লাসও হয়, তবে এ পোস্টে সেটা আলোচনা করবো না।] তো ধাপে ধাপে শুরু করা যাক। আমি ধরে নিচ্ছি ভ্যারিয়েবলের স্কোপ রেজ্যুলেশন সম্পর্কে আমাদের মোটামুটি ভাল ধারনা আছে, তাই এ সম্পর্কে আর বাড়তি কিছু লিখলাম না।

### ফাংশনের কারিকুরি

এই কোড ব্লক টা দেখি:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | def hello():  print("Hello World!")  hello()  # output: Hello World!  hi = hello # interesting!  print(hi)  # output: <function hello at 0x7ff330b412f0>  hi()  # output: Hello World! |

8 নম্বর লাইন থেকে কোডটা ইন্টারেস্টিং হওয়া শুরু করেছে। আমরা জানি পাইথনে সব কিছুই এক একটা অবজেক্ট। ফাংশনও। এই লাইনে আমরা hello কে hi তে এসাইন করেছি। লক্ষনীয়, এখানে hello এর পাশে () (ব্র্যাকেট/প্যারেন্থেসিস) দেই নি। অর্থাৎ এখানে hello ফাংশনটি এক্সিকিউট বা কল হয় নি। 10 নম্বর লাইনের আউটপুট দেখলে ব্যাপারটা আরো পরিষ্কার হবে। আর 13 নম্বর লাইনে hi কে কল করা হয়েছে, আউটপুট পেয়েছি ঠিক hello এর মত।

### ফাংশনের ভেতর ফাংশন!

হ্যা, পাইথনে আমরা ফাংশনের ভেতর ফাংশন ডিফাইন করতে পারি। অন্যভাবে বললে আমরা নেস্টেড ফাংশন বানাতে পারি। এরকম:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | def hello():  print("Inside hello")  def nested():  print("Inside nested")  nested()  print("Inside hello again")  hello() |

আউটপুট:

Inside hello

Inside nested

Inside hello again

আরেকটি কোড ব্লক দেখি:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | def count(value):  def increment():  return value + 1  print("Incremented value of {} is {}".format(value, increment()))  count(5)  # output: Incremented value of 5 is 6 |

শুরুতেই লিখেছিলাম ভ্যারিয়েবলের স্কোপ রেজ্যুলেশন সম্পর্কে লিখবো না। কোডটা একটু ভাল মত লক্ষ্য করলেই আশা করি বুঝতে পারবেন।

### ফাংশন থেকে ফাংশন রিটার্ন!

ফাংশন থেকে ইচ্ছা করলে আমরা ফাংশন রিটার্নও করতে পারি! এই কোডটা দেখলে ব্যাপারটা পরিষ্কার হবে:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | def hello():  def nested():  print("Inside nested")  return nested  hi = hello()  print(hi)  # output: <function hello.<locals>.nested at 0x7f1ed4e94488>  hi()  # output: Inside nested |

9 নম্বর লাইনে hello() কে কল করায় এটা nested কে রিটার্ন করেছে, তা এসাইন হয়েছে hi তে। পূর্বের কোড ব্লক গুলো ফলো করলে এটি সহজেই বোঝা যাবে।

আরেকটি কোড ব্লক দেখি:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | def count(value):  def increment(n):  return value + n  return increment  counter = count(5)  print(counter(1)) # Guess the output! |

কি, আউটপুট গেস করেছেন? এটা আর এক্সপ্লেইন করবো না। আউটপুট হবে 6।

### ফাংশনের আর্গুমেন্ট/প্যারামিটার হিসেবে ফাংশন

সরাসরি একটা কোড স্নিপেট দেখে ফেলি:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | def hello():  print("Hello World!")  def hi(func):  print("Hi!")  func()  hi(hello) |

এখানে hi ফাংশনের প্যারামিটার হিসেবে hello কে পাস করা হয়েছে। hi এর ভেতর hello কল হয়েছে। আউটপুট হবে এরকম:

Hi!

Hello World!

## **ডেকরেটর**

এখন হচ্ছে মূল বিষয়, ডেকরেটর। আমরা আগেই জেনেছি, ডেকরেটর হচ্ছে এমন ফাংশন যা অন্য ফাংশনের ফাংশনালিটি মডিফাই করে। এখন তাহলে একটু সাজানো গুছানো উদাহরণ দেখে নেই:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | def mydecorator(func):  def wrapper():  print("Before calling func()")  func()  print("After calling func()")  return wrapper  def hello():  print("Hello World!")  hello = mydecorator(hello)  hello() |

এখানে mydecorator ফাংশনটি প্যারামিটার হিসেবে আরেকটি ফাংশন এক্সপেক্ট করছে। এর মধ্যকার wrapper ফাংশনটিতে প্যারামিটারে পাওয়া ফাংশন কল করার আগে এবং পরে কিছু কাজ হচ্ছে। আর mydecorator থেকে wrapper কে রিটার্ন করা হচ্ছে। ১৫ নম্বর লাইনে mydecorator কে hello প্যারামিটার দিয়ে কল করা হয়েছে। রিটার্ন ভ্যালু এসাইন করা হয়েছে আবার hello তে। অর্থাৎ mydecorator এর মাধ্যমে hello মডিফাই হয়েছে। [প্রয়োজনে আবার খেয়াল করুন।] সব শেষ লাইনে hello() কল হয়েছে। আউটপুট হবে এরকম:

Before calling func()

Hello World!

After calling func()

নাম এবং কাজ দেখে বোঝাই যাচ্ছে mydecorator হচ্ছে আমাদের কাঙ্খিত সেই ডেকরেটর।

তবে ডেকরেটর ব্যাবহারের সুন্দর একটি সিনট্যাক্স আছে, @। উপরের কোড কে আমরা সুন্দর করে এভাবে লিখতে পারি:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | def mydecorator(func):  def wrapper():  print("Before calling func()")  func()  print("After calling func()")  return wrapper  @mydecorator  def hello():  print("Hello World!")  hello() |

অর্থাৎ, hello = mydecorator(hello) এই লাইনের পরিবর্তে আমরা hello ফাংশনটি ডিফাইনের ঠিক আগে @mydecorator লিখেছি। পূর্বের মত একই কাজ হবে।

### বাস্তব উদাহরণ

এবার একটি বাস্তব উদাহরণ দেখা যাক। মনে করি আমাদের একটি ফাংশন আছে, আমরা চাই যখন এটি কল হবে ঠিক ওই সময় যেন লগ হিসবে একটা ফাইলে থাকে। এর সমাধান দেখে নেয়া যাক:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | from datetime import datetime  from time import sleep  def log(func):  def wrapper():  with open('log.txt', 'a') as file:  now = datetime.now().strftime("%I:%M:%S%p on %d %B, %Y")  file.write(now + '\n')  func()  return wrapper  @log  def hello():  print('Hello World!')  for i in range(3):  hello()  sleep(5) |

এই প্রোগ্রাম রান করলে কারেন্ট ওয়ার্কিং ডিরেক্টরিতে log.txt নামের একটা ফাইল তৈরি হবে, সেটি খুললে hello() এক্সিকিউট হওয়ার সময় গুলো পাওয়া যাবে।

### প্যারামিটার/আর্গুমেন্ট সহ ডেকরেটর

যদি log ডেকরেটর টায় প্যারামিটার হিসেবে ফাইলের নাম দিয়ে দেয়া যেত, log.txt এর পরিবর্তে আমাদের প্রয়োজন মত নাম, তাহলে সুবিধা হতো না? হ্যা, ডেকরেটরে প্যারামিটার/আর্গুমেন্ট পাস করা সম্ভব। এজন্য আমাদের ডেকরেটরকে আরেকটা ফাংশনের মধ্যে নেস্টেড আকারে রাখা লাগবে। এরকম:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | from datetime import datetime  from time import sleep  def log(filename='log.txt'):  def log\_decorator(func):  def wrapper():  with open(filename, 'a') as file:  now = datetime.now().strftime("%I:%M:%S%p on %d %B, %Y")  file.write(now + '\n')  func()  return wrapper  return log\_decorator  @log(filename='history.txt')  def hello():  print('Hello World!')  for i in range(3):  hello()  sleep(5) |

এখন আমাদের লগ history.txt খুললে পাওয়া যাবে। আর যদি @log এ কোন প্যারামিটার পাস না করা হয়, তাহলে ডিফল্ট ভাবে log.txt তে লগ থাকবে।

এইত! এই ছিল পাইথনের ডেকরেটর‍্স‍ কনসেপ্ট। যদিও যেভাবে উপস্থাপন করতে চেয়েছিলাম সেভাবে পারি নি, তারপরও আশা করছি অপেক্ষাকৃত নতুনেরা উপকৃত হবে। সামনে কোন এক সময় ক্লাস ডেকরেটর নিয়ে লিখবো ইনশাআল্লাহ।

#### ডেকরেটর এর ব্যবহার

# **ফাংশন চাপ্টারে আমরা ফাংশন নিয়ে অল্প-বিস্তর ব্যাপক জ্ঞান লাভ করেছি। পাইথনে ফাংশন হল ফার্স্ট-ক্লাস অবজেক্ট। এর মানে হল আমরা ফাংশনকে নরমাল ভ্যালু বা ভ্যারিয়েবলের মত করে পাস করতে পারি, প্যারামিটার বা আর্গুমেন্ট হিসাবে ব্যবহার করতে পারি। বুঝলাম না মনে হয়। একটা উদাহরণ দেখা যাক।**

def get\_int\_as\_str(number):

return str(number)

def print\_int(my\_function, number):

print(my\_function(number))

return

print\_int(get\_int\_as\_str, 130)

**আউটপুট**

130

এখানে আমরা print\_int() ফাংশনের ভিতর get\_int\_as\_str() ফাংশনকে আর্গুমেন্ট হিসাবে পাস করেছি। পরে সেটা দিয়ে কোডিং কারিশমা দেখাইছি। এ তো গেল ফাংশনকে আর্গুমেন্ট হিসাবে পাস করার কথা। যদি ফাংশনকেই রিটার্ন করতে চাই? এও কি সম্ভব! হুম, সব সম্ভবের পাইথনে সবই সম্ভব।

def get\_int\_as\_str(number):

print(str(number))

return

def print\_int(my\_function, number):

return my\_function(number)

print\_int(get\_int\_as\_str, 130)

**আউটপুট**

130

এখানে আমরা print\_int() ফাংশনের ভিতর get\_int\_as\_str() ফাংশনকে ও একটা ইন্টিজার ভ্যালুকে আর্গুমেন্ট হিসাবে পাস করেছি। print\_int() ফাংশনের রিটার্ন অংশে আবার get\_int\_as\_str() ফাংশনকে রিটার্ন করেছি। এইক্ষেত্রে একটা কাহিনি আছে। get\_int\_as\_str() ফাংশনটা রিটার্ন হবার সময় এক্সিকিউট হওয়া শুরু করে। তাই দিনশেষে get\_int\_as\_str() ফাংশন যা রিটার্ন করে print\_int() ফাংশনও তাই রিটার্ন করে।

আমরা কিন্তু চাইলে get\_int\_as\_str() ফাংশনটাকে print\_int() ফাংশনের ভিতরেই ডিফাইন করতে পারতাম।

def print\_int(number):

def get\_int\_as\_str(number):

print(str(number))

return

get\_int\_as\_str(number)

return

print\_int(130)

**আউটপুট**

130

উপরের প্রোগ্রামটাকে আরেকটু মডিফাই করে নিচের চেহারা দিতে পারি আমরা।

def print\_int(number):

def get\_int\_as\_str(number):

print(str(number))

return

return get\_int\_as\_str(number)

print\_int(130)

**আউটপুট**

130

নিজেদের অজান্তেই কিন্তু আমরা একটা ডেকোরেটর লিখে ফেলেছি। এইক্ষেত্রে print\_int() হল ডেকোরেটর। পাইথনের ভাষায়, ডেকোরেটর হল এমন একটা ফাংশন যা অন্য কোন ফাংশনের কার্যপরিধি কোন প্রকার মডিফিকেশন ছাড়াই এক্সটেন্ড করে। ডেকোরেটর লেখার সিনট্যাক্স হল @decorator\_name। উপরের প্রোগ্রামটাকে এখন আমরা এভাবে লিখতে পারি।

def print\_int(my\_function):

def any\_function():

return my\_function

return any\_function()

@print\_int

def get\_int\_as\_str(number):

print(str(number))

return

get\_int\_as\_str(130)

**আউটপুট**

130

বুঝলাম সবাই? ডেকোরেটর তো বুঝলাম। কিন্তু এর সুবিধা তো বোধগম্য হল না। এরজন্য আমরা আরেকটা প্রোগ্রাম দেখব। আমরা এখন একটা ডেকোরেটর ফাংশন লিখব যেটা কোন ফাংশন এক্সিকিউট হতে কত সময় নেয় তা জাতিরে জানান দিবে। এইক্ষেত্রে পাইথন স্টান্ডার্ড মডিউল time এর time() ফাংশনের সহায়তা নেব আমরা।

from time import time

def timer(any\_function):

def count\_time():

start = time()

any\_function()

stop = time()

print(stop-start, 'seconds')

return

return count\_time

@timer

def hello():

print('Hello World!')

return

@timer

def another\_function():

for item in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]:

print(item)

return

hello()

another\_function()

**আউটপুট**

Hello World!

0.006817340850830078 seconds

1

2

3

4

5

6

7

8

9

10

0.041167497634887695 seconds

## **বিল্ট-ইন ডেকোরেটর**

কাস্টম ডেকোরেটর নিয়ে বেশ আলোচনা হয়েছে। এবার আমরা পাইথনের শক্তিশালী তিনটি বিল্ট-ইন ডেকোরেটর সম্পর্কে জানব।

### ক্লাস মেথড - @classmethod

এই ডেকোরেটরটা সম্পর্কে জানার আগে আমরা এর একটা প্রয়োগ দেখব:

class MyClass:

"""Simple Class to define something"""

def \_\_init\_\_(self):

pass

def square(self, x):

return x\*\*2

@classmethod

def cube(self, x):

return x\*\*3

if \_\_name\_\_ == "\_\_main\_\_":

myclass = MyClass()

print(myclass.square(3))

print(myclass.cube(3))

print(MyClass.cube(3))

print(MyClass.square(3))

**আউটপুট**

9

27

27

Traceback (most recent call last):

File "/home/maateen/Desktop/test.py", line 19, in <module>

print(MyClass.square(3))

TypeError: square() missing 1 required positional argument: 'x'

আমরা একটা নিরীহ গোবেচারা ধরনের ক্লাস লিখেছি। এই ক্লাসে square() এবং cube() নামের দুটি মেথডও লিখেছি। আর বিশেষায়িত দিকটা হলো cube () মেথডে ক্লাসমেথড ডেকোরেটর ব্যবহার করেছি। আউটপুটে আমরা TypeError দেখতে পাচ্ছি। square() ফাংশনের একটা পজিশনাল আর্গুমেন্ট মিসিং। কাহিনিটা ময়নাতদন্ত করা যাক।

১৫ নম্বর লাইনে আমরা MyClass-এর একটা ইনস্ট্যান্স বা অবজেক্ট তৈরি করেছি এবং ১৬ ও ১৭ নম্বর লাইনে এই অবজেক্টটা ব্যবহার করে মেথড দুটো কল করেছি। আর তারা ঠিকমতো কাজও করেছে। ১৮ নম্বর লাইনে আমরা সরাসরি ক্লাসের নাম ধরে মেথড কল করেছি। এবারও কোনো সমস্যা হয়নি। কিন্তু ১৯ নম্বর লাইনে একই কাজ করতে গিয়ে এরর দেখাল। ঘটনা কী?

ঘটনা হলো, cube() মেথডে আমরা ক্লাসমেথড ডেকোরেটর ব্যবহার করেছি কিন্তু square() মেথডে তা করিনি। যখন ক্লাসমেথড ডেকোরেটর ব্যবহার করা হয় তখন ওই ফাংশন অবজেক্ট তৈরি না করেও কল করা যায়।

### স্ট্যাটিক মেথড - @staticmethod

সবার আগে আমরা স্ট্যাটিকমেথড ডেকোরেটরের একটা প্রয়োগ দেখব :

class MyClass:

"""Simple Class to define something"""

def \_\_init\_\_(self):

pass

def square(self, x):

return x\*\*2

@staticmethod

def cube(x):

return x\*\*3

if \_\_name\_\_ == "\_\_main\_\_":

myclass = MyClass()

print(myclass.square(3))

print(myclass.cube(3))

print(MyClass.cube(3))

print(MyClass.square(3))

**আউটপুট**

9

27

27

Traceback (most recent call last):

File "test.py", line 19, in <module>

print(MyClass.square(3))

TypeError: square() missing 1 required positional argument: 'x'

ঘটনা কিন্তু আগেরটাই। আর ঘটনার কারণও সেই আগেরটাই। তাহলে আমাদের মনে দুটি প্রশ্ন উঠতে পারে। প্রথমত, cube() মেথডে প্রথম আর্গুমেন্ট হিসেবে self ব্যবহার করা হয়নি কেন? দ্বিতীয়ত, দুটি ডেকোরেটরের মধ্যে তাহলে পার্থক্য কী?

স্ট্যাটিক মেথড ডেকোরেটরের ক্ষেত্রে মেথডের প্রথম আর্গুমেন্ট ক্লাস ইনস্ট্যান্স বা self হবে না। সাধারণ ফাংশন লেখার মতো করে লিখতে হবে। ক্লাসমেথড ডেকোরেটরের ক্ষেত্রে প্রথম আর্গুমেন্ট হবে ক্লাস বা ক্লাস ইনস্ট্যান্স। দুটোর মধ্যে এটাই হলো অমিল। আর দুজনের কাজই কিন্তু এক। একটা মেথডকে সরাসরি ক্লাস থেকে কল করতে পারা যায়। এই কাজই কিন্তু আমরা ওপরে করেছি।

### প্রোপার্টি - @property

পাইথনের অস্থির কনসেপ্টগুলোর মধ্যে প্রোপার্টি একটা। এই কনসেপ্ট অবজেক্ট ওরিয়েন্টেড প্রোগ্রামিংকে করেছে আরও পোক্ত। যাহোক, পাইথন শেলে একটা প্রোগ্রাম লিখি:

>>> class Weather:

...

... def \_\_init\_\_(self, temperature=0):

... self.temperature = temperature

...

... def to\_fahrenheit(self):

... return self.temperature \* 1.8

...

>>> weather = Weather()

>>> weather.temperature

0

>>> weather.temperature =25

>>> weather.temperature

25

>>> weather.to\_fahrenheit()

45.0

এখানে আমরা ক্লাসের ইনস্ট্যান্স থেকে এর প্রোপার্টি অ্যাক্সেস করেছি। এবার প্রোপার্টি ডেকোরেটর ব্যবহার করে আমরা একটা উদাহরণ দেখব :

class MyClass:

"""Simple Class to define something"""

def \_\_init\_\_(self, first\_name, last\_name):

self.first\_name = first\_name

self.last\_name = last\_name

@property

def full\_name(self):

return self.first\_name + ' ' + self.last\_name

if \_\_name\_\_ == "\_\_main\_\_":

myclass = MyClass('Maksudur', 'Rahman')

print(myclass.full\_name)

myclass.full\_name = 'New Name'

**আউটপুট**

Maksudur Rahman

Traceback (most recent call last):

File "test.py", line 15, in <module>

myclass.full\_name = 'New Name'

AttributeError: can't set attribute

প্রোপার্টি ডেকোরটের ব্যবহার করে আমরা full\_name() মেথডকে রিড-অনলি অ্যাট্রিবিউটে পরিণত করেছি। তাই সাধারণভাবে ডট চিহ্ন (.) দিয়ে full\_name-কে অ্যাক্সেস করতে পারছি। আর রিড-অনলি হওয়ায় এই অ্যাট্রিবিউটে নতুন কিছু দেওয়া যাচ্ছে না।

আশা করি বুঝেছি সবাই। তাহলে এরকম আরো কিছু ডেকোরেটর লিখে প্রাকটিস চালিয়ে যাই।

#### এটারেটর

## **লুপ চাপ্টারে for লুপের ভিতরে আমরা ইটারেটরের হালকা-পাতলা প্রয়োগ দেখেছিলাম। আমরা দেখেছিলাম, for লুপ দিয়ে লিস্ট, টাপল, ডিকশনারি, স্ট্রিং এসব অবজেক্টকে ইটারেট করা যায়। ইটারেট করা যায় বলে এদেরকে ইটারেবল অবজেক্ট বলে। বিল্ট-ইন ফাংশন iter() এর ভিতর যদি আমরা কেন ইটারেবল অবজেক্টকে পাস করি তবে তা একটা ইটারেটর রিটার্ন করে। আর \_\_next\_\_() মেথড দিয়ে আমরা একটা ইটারেটরের পরবর্তী এলিমেন্ট বা আইটেমকে অ্যাক্সেস করতে পারি। সবগুলো আইটেম অ্যাক্সেস করতে করতে যখন আর কোন আইটেম বাকি থাকে না তখন পাইথন StopIteration এরর থ্রো করে। একটা উদাহরণ দেখা যাক।**

>>> x = iter([1, 2, 3])

>>> x

<list\_iterator object at 0x7f0f1694b1d0>

>>> x.\_\_next\_\_()

1

>>> x.\_\_next\_\_()

2

>>> x.\_\_next\_\_()

3

>>> x.\_\_next\_\_()

Traceback (most recent call last):

File "<pyshell#6>", line 1, in <module>

x.\_\_next\_\_()

StopIteration

আমরা এখন একটা ইটারেটর ক্লাস পয়দা করব। আমাদের এই ইটারেটর বিল্ট-ইন range() ফাংশনের মত কাজ করবে, তবে রিভার্সলি। মানে range(5) মানে তো 0, 1, 2, 3, 4। কিন্তু আমাদের তৈরি ইটারেটর ক্লাস এটাকে 5, 4, 3, 2, 1, 0 এরকম কিছু একটা বানাবে। তো চেষ্টা করা যাক।

class revrange:

def \_\_init\_\_(self, n):

self.n = n

self.i = n

def \_\_iter\_\_(self):

return self

def \_\_next\_\_(self):

if self.n >= 0:

if self.i == self.n:

self.n = self.n - 1

return self.i

else:

self.i = self.n

self.n = self.n - 1

return self.i

else:

raise StopIteration

for temp in revrange(5):

print(temp)

**আউটপুট**

5

4

3

2

1

0

\_\_iter\_\_() মেথডটা কিন্তু খুব গুরুত্বপূর্ণ এখানে। এটা ছাড়া পুরো ক্লাসটাই অচল হয়ে থাকবে।

**জেনারেটর**

একটি [জেনারেটর](http://www.python.org/dev/peps/pep-0255/" \t "_blank) হ'ল এমন একটি ফাংশন যা কোনও অবজেক্টকে কল করে যার উপর আপনি কল করতে পারেন next, যেমন প্রতিটি কলের জন্য এটি কিছু মান দেয়, যতক্ষণ না এটি StopIterationব্যতিক্রম উত্থাপন করে , যা সংকেত দেয় যে সমস্ত মান উত্পন্ন হয়েছে। এ জাতীয় অবজেক্টকে *পুনরুক্তি* বলা হয় ।

সাধারণ ফাংশনগুলি returnজাভা হিসাবে যেমন ব্যবহার করে একটি একক মান প্রদান করে । পাইথনে অবশ্য একটি বিকল্প রয়েছে, যাকে বলে yield। yieldকোনও ফাংশনে কোথাও ব্যবহার করা এটিকে জেনারেটর করে তোলে। এই কোডটি পর্যবেক্ষণ করুন:

>>> def myGen(n):

... yield n

... yield n + 1

...

>>> g = myGen(6)

>>> next(g)

6

>>> next(g)

7

>>> next(g)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

StopIteration

যেমন আপনি দেখতে পাচ্ছেন, myGen(n)এটি একটি ফাংশন যা ফল দেয় nএবং n + 1। [next](http://docs.python.org/3/library/functions.html" \l "next" \t "_blank)সমস্ত মান প্রাপ্ত না হওয়া পর্যন্ত প্রতিটি কল একক মান দেয় yield forলুপগুলি nextব্যাকগ্রাউন্ডে কল করে,

>>> for n in myGen(6):

... print(n)

...

6

7

তেমনিভাবে *[জেনারেটর এক্সপ্রেশন রয়েছে](http://www.python.org/dev/peps/pep-0289/" \t "_blank)* যা কিছু সাধারণ জেনারেটরের সংক্ষিপ্তভাবে বর্ণনা করার একটি উপায় সরবরাহ করে:

>>> g = (n for n in range(3, 5))

>>> next(g)

3

>>> next(g)

4

>>> next(g)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

StopIteration

নোট করুন যে জেনারেটর এক্সপ্রেশনগুলি অনেকগুলি *[তালিকা বোঝার](http://docs.python.org/3/tutorial/datastructures.html" \l "list-comprehensions" \t "_blank)* মতো :

>>> lc = [n for n in range(3, 5)]

>>> lc

[3, 4]

লক্ষ্য করুন যে কোনও জেনারেটর অবজেক্ট *একবার* তৈরি হয়েছিল তবে এর কোডটি একবারে চালানো হয় *না* । কেবলমাত্র nextকোডটি বাস্তবায়নের জন্য (অংশের) কল । কোনও জেনারেটরে কোড সম্পাদন বন্ধ হয়ে যায় যখন কোনও yieldবিবৃতি পৌঁছে যায়, যার উপর এটি একটি মান দেয়। nextতারপরে পরবর্তী কলের ফলে জেনারেটরটি শেষের পরে যে রাজ্যে রেখেছিল সেই স্থানেই মৃত্যুদন্ড কার্যকর করা যায় yield। এটি নিয়মিত ক্রিয়াকলাপগুলির সাথে একটি মৌলিক পার্থক্য: যারা সর্বদা "শীর্ষে" সম্পাদন শুরু করে এবং কোনও মান ফেরত দেওয়ার পরে তাদের রাষ্ট্রটি বাতিল করে দেয়।

এই বিষয় সম্পর্কে আরও কিছু বলা দরকার। এটি উদাহরণস্বরূপ sendকোনও জেনারেটরে ( [রেফারেন্স](http://docs.python.org/3/reference/expressions.html" \l "yield-expressions" \t "_blank) ) ডেটা পাওয়া সম্ভব । তবে এটি এমন একটি বিষয় যা আমি আপনাকে জেনারেটরের প্রাথমিক ধারণাটি না বোঝা পর্যন্ত ততক্ষণ সন্ধান করার পরামর্শ দিই না।

এখন আপনি জিজ্ঞাসা করতে পারেন: কেন জেনারেটর ব্যবহার? বেশ কয়েকটি ভাল কারণ রয়েছে:

* কয়েকটি ধারণা জেনারেটর ব্যবহার করে আরও বেশি সংবেদনশীলভাবে বর্ণনা করা যায়।
* মানগুলির তালিকায় ফিরিয়ে দেয় এমন একটি ফাংশন তৈরির পরিবর্তে, একটি জেনারেটর লিখতে পারে যা উড়ে মানগুলি উত্পন্ন করে। এর অর্থ হ'ল কোনও তালিকা তৈরি করার দরকার নেই, যার ফলে ফলাফলটি আরও মেমরির দক্ষ। এইভাবে কেউ এমন কোনও ডেটা স্ট্রিমগুলি বর্ণনা করতে পারে যা মেমরির সাথে ফিট করার পক্ষে খুব বড় large
* জেনারেটর *অসীম* স্ট্রিমগুলি বর্ণনা করার জন্য একটি প্রাকৃতিক উপায়ে অনুমতি দেয় । উদাহরণস্বরূপ [ফিবোনাচি সংখ্যাগুলি](http://en.wikipedia.org/wiki/Fibonacci_number" \t "_blank) বিবেচনা করুন :
* >>> def fib():
* ... a, b = 0, 1
* ... while True:
* ... yield a
* ... a, b = b, a + b
* ...
* >>> import itertools
* >>> list(itertools.islice(fib(), 10))
* [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]

এই কোডটি [itertools.islice](http://docs.python.org/3/library/itertools.html" \l "itertools.islice" \t "_blank)অসীম স্ট্রিম থেকে সীমাবদ্ধ সংখ্যক উপাদান গ্রহণ করতে ব্যবহার করে। আপনাকে [itertools](http://docs.python.org/3/library/itertools.html" \t "_blank)মডিউলটিতে ফাংশনগুলি ভালভাবে দেখার পরামর্শ দেওয়া হচ্ছে , কারণ এগুলি খুব সহজেই উন্নত জেনারেটর লেখার জন্য প্রয়োজনীয় সরঞ্জাম।

  Py **পাইথন সম্পর্কে <= 2.6:** উপরের উদাহরণগুলিতে nextএকটি ফাংশন যা \_\_next\_\_প্রদত্ত বস্তুটিতে পদ্ধতিটিকে কল করে । পাইথনে <= 2.6 এ o.next()পরিবর্তে কিছুটা আলাদা কৌশল ব্যবহার করা হয়েছে next(o)। পাইথন ২.7 এর next()কল রয়েছে .nextযাতে আপনার নীচের ২.7 ব্যবহার করার দরকার নেই:

>>> g = (n for n in range(3, 5))

>>> g.next()

3

একটি জেনারেটর কার্যকরভাবে একটি ফাংশন যা এটি শেষ হওয়ার পূর্বে (ডেটা) ফেরত দেয় তবে এটি এই মুহুর্তে বিরতি দেয় এবং আপনি সেই মুহুর্তে ফাংশনটি আবার শুরু করতে পারেন।

>>> def myGenerator():

... yield 'These'

... yield 'words'

... yield 'come'

... yield 'one'

... yield 'at'

... yield 'a'

... yield 'time'

>>> myGeneratorInstance = myGenerator()

>>> next(myGeneratorInstance)

These

>>> next(myGeneratorInstance)

words

ইত্যাদি। জেনারেটরগুলির (বা এক) উপকারটি হ'ল যেহেতু তারা একবারে ডেটার সাথে এক টুকরো লেনদেন করে, আপনি প্রচুর পরিমাণে ডেটা মোকাবেলা করতে পারেন; তালিকাগুলি সহ অতিরিক্ত মেমরির প্রয়োজনীয়তা একটি সমস্যা হয়ে উঠতে পারে। জেনারেটর, ঠিক তালিকার মতো, পুনরাবৃত্তিযোগ্য, তাই সেগুলি একই পদ্ধতিতে ব্যবহার করা যেতে পারে:

>>> for word in myGeneratorInstance:

... print word

These

words

come

one

at

a

time

নোট করুন যে জেনারেটর উদাহরণস্বরূপ অসীম মোকাবিলার জন্য আরেকটি উপায় সরবরাহ করে

>>> from time import gmtime, strftime

>>> def myGen():

... while True:

... yield strftime("%a, %d %b %Y %H:%M:%S +0000", gmtime())

>>> myGeneratorInstance = myGen()

>>> next(myGeneratorInstance)

Thu, 28 Jun 2001 14:17:15 +0000

>>> next(myGeneratorInstance)

Thu, 28 Jun 2001 14:18:02 +0000

জেনারেটর একটি অসীম লুপকে আবদ্ধ করে, তবে এটি কোনও সমস্যা নয় কারণ আপনি প্রতিবার যখনই জিজ্ঞাসা করবেন কেবল তখনই প্রতিটি উত্তর পান।

30

প্রথমত, **জেনারেটর** শব্দটি মূলত পাইথনে কিছুটা অশুভ সংজ্ঞায়িত হয়েছিল, যার ফলে প্রচুর বিভ্রান্তি হয়েছিল। আপনি সম্ভবত গড় **iterators** এবং **iterables** (দেখুন [এখানে](http://docs.python.org/library/stdtypes.html" \l "typeiter" \t "_blank) )। তারপরে পাইথনে **জেনারেটর ফাংশন** (যা একটি জেনারেটরের বস্তু ফেরত দেয়), **জেনারেটর অবজেক্টস** (যা পুনরুক্তি করা হয়) এবং **জেনারেটর এক্সপ্রেশন** (যা **জেনারেটরের** সাথে মূল্যায়ন করা হয়) রয়েছে।

মতে [জন্য শব্দকোষ এন্ট্রি](http://docs.python.org/glossary.html" \l "term-generator" \t "_blank)*[জেনারেটরের](http://docs.python.org/glossary.html" \l "term-generator" \t "_blank)* মনে হচ্ছে যে সরকারী পরিভাষা এখন যে **জেনারেটরের** "উত্পাদক ফাংশন" জন্য ছোট। অতীতে ডকুমেন্টেশন শর্তাবলীকে বেমানানভাবে সংজ্ঞায়িত করেছিল, তবে ভাগ্যক্রমে এটি ঠিক করা হয়েছে।

যথাযথ হওয়া এবং আরও নির্দিষ্টকরণ ছাড়াই "জেনারেটর" শব্দটি এড়ানো এখনও ভাল ধারণা হতে পারে।

2

হুম আমি মনে করি আপনি ঠিক বলেছেন, কমপক্ষে পাইথন ২.6 এর কয়েকটি লাইনের পরীক্ষা অনুযায়ী। একটি জেনারেটর এক্সপ্রেশন একটি পুনরুক্তি (ওরফে 'জেনারেটর অবজেক্ট') দেয়, জেনারেটর নয়।

22

একটি জেনারেটর তৈরির জন্য জেনারেটরগুলি শর্টহ্যান্ড হিসাবে বিবেচনা করা যেতে পারে। তারা জাভা আইট্রেটারের মতো আচরণ করে। উদাহরণ:

>>> g = (x for x in range(10))

>>> g

<generator object <genexpr> at 0x7fac1c1e6aa0>

>>> g.next()

0

>>> g.next()

1

>>> g.next()

2

>>> list(g) # force iterating the rest

[3, 4, 5, 6, 7, 8, 9]

>>> g.next() # iterator is at the end; calling next again will throw

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

StopIteration

আশা করি এটি আপনার সন্ধান করছে / এটিকে সহায়তা করে।

**হালনাগাদ:**

অন্যান্য অনেক উত্তর প্রদর্শিত হচ্ছে, জেনারেটর তৈরির বিভিন্ন উপায় রয়েছে। উপরের উদাহরণ হিসাবে আপনি প্রথম বন্ধনী বাক্য গঠন ব্যবহার করতে পারেন, বা আপনি ফলন ব্যবহার করতে পারেন। আর একটি আকর্ষণীয় বৈশিষ্ট্য হ'ল জেনারেটরগুলি "অসীম" হতে পারে - পুনরাবৃত্তিকারীরা থামে না:

>>> def infinite\_gen():

... n = 0

... while True:

... yield n

... n = n + 1

...

>>> g = infinite\_gen()

>>> g.next()

0

>>> g.next()

1

>>> g.next()

2

>>> g.next()

3

...

1

এখন, জাভা এর রয়েছে Streamযা জেনারেটরের সাথে অনেক বেশি সাদৃশ্যযুক্ত, আপনি সম্ভবত দৃশ্যমান একটি বিস্ময়কর পরিমাণ ঝামেলা ছাড়াই পরবর্তী উপাদানটি পেতে পারেন না।

12

জাভা সমতুল্য নেই।

এখানে কিছুটা নিবিড় উদাহরণ রয়েছে:

#! /usr/bin/python

def mygen(n):

x = 0

while x < n:

x = x + 1

if x % 3 == 0:

yield x

for a in mygen(100):

print a

জেনারেটরে একটি লুপ থাকে যা 0 থেকে এন পর্যন্ত চলে এবং লুপ ভেরিয়েবলটি 3 এর একাধিক হয়, এটি চলকটি দেয়।

forলুপের প্রতিটি পুনরাবৃত্তির সময় জেনারেটর কার্যকর করা হয়। এটি যদি প্রথমবারের মতো উত্পাদক কার্যকর করে থাকে তবে এটি শুরুতে শুরু হয়, অন্যথায় এটি পূর্ববর্তী সময় থেকে ফলন করেছিল।

#### ল্যামডা ফাঙ্শন

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

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

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

উদাহরণ –

>>> def my\_function(func, arg):

... return func(arg)

...

>>> print(my\_function(lambda x: 2 \* x, 5))

10

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

আরেকটি উদাহরণ,

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

8

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

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

**রেকারশন**

কোন কিছু যদি নিজেকে নিজে পুনরায় ডাকে, করে তাই হচ্ছে **[রিকারশন/ Recursion।](https://en.wikipedia.org/wiki/Recursion_(computer_science))** যেমন একটা ছবির মধ্যে ঐ ছবিটি, ঐ ছবিটির ভেতরে আবার ঐ ছবিটি… এ প্রসেসটা হচ্ছে রিকারশনের একটা উদারহণ।

একটা খালি মাঠে গিয়ে নিজেকে নিজে ডাকলে বা একটা বিশাল হল রুমে নিজের নাম ধরে ডাকা ডাকি করলে রিকার্শন অনুভব করা যাবে।

নিচের ছবিটা দেখি, এটাও রিকারশন।

গুগলে Recursion লিখে সার্চ করে দেখুন। বার বার লেখা উঠবে Did you mean: Recursion। বানান ঠিক থাকার পর ও এটা দেখাবে। গুগল মজা করে একটা রিকারশন বসিয়ে দিয়েছে Recursion সার্চ টার্ম এর উপর।

**Recursive Algorithm** হচ্ছে যে অ্যালগরিদম নিজেকে নিজে কল করে, তা। কম্পিউটার প্রোগ্রামিং এ  রিকারসিভ অ্যালগরিদম ব্যবহার করে কোন প্রোগ্রামে রিকারশন ব্যবহার করা হয়। বিভিন্ন প্রোগ্রামিং ল্যাঙ্গুয়েজে একটি ফাংশন নিজেকে কল করার মাধ্যমে রিকারশন এর প্রয়োগ করা হয়। আমরা রিকার্সিভ ফাংশন ব্যবহার করে দুই একটা রিকার্সিভ অ্যালগরিদম দেখব। আর প্রোগ্রামিং ল্যাঙ্গুয়েজ হিসেবে ব্যবহার করব সি।

রিকার্সিভ ফাংশন কি তা তো এখন সহজেই বলা যাচ্ছে তাই না? যে ফাংশন নিজেকে নিজে কল করে, তাই হচ্ছে রিকার্সিভ ফাংশন। ফাংশন সম্পর্কে ধারণা না থাকলে এ লেখাটা দেখা যেতে পারেঃ**[সি তে ফাংশন](https://jakir.me/c-functions/)**

রিকারশনের সুবিধে হচ্ছে কোড সহজ করে, অনেক গুলো কোড লেখার পরিবর্তে কয়েক লাইনের কোড দিয়ে একটা সমস্যা সমাধান করা যায়।

***void f() {***  
***f() …***  
***}***

এটা একটা রিকার্সিভ ফাংশন, কারণ ফাংশনটি নিজেকে নিজে কল করেছে। এটাকে বলে সরাসরি কল। আবার ফাংশন সরাসরি কল না করেও এমন একটা ফাংশনকে কল করতে পারে, যে ফাংশনটি প্রথম ফাংশনকে কল করে। নিচের উদারহণটি দেখিঃ

***void f() {***  
***g() …***  
***}***

***void g() {***  
***f() …***  
***}***

f ফাংশনটি g ফাংশনকে কল করেছে। আবার g ফাংশন f ফাংশনকে কল করেছে। এটাও রিকার্সিভ ফাংশন।

আমরা একটা রিকার্সিভ ফাংশন লিখি, যেটা ইউজার থেকে একটা নাম্বার নিবে, তারপর ঐ সংখ্যা থেকে এর পর ১ থেকে ঐ সংখ্যা পর্যন্ত সকল সংখ্যা প্রিন্ট করবে। এটা সিম্পল একটা প্রোগ্রাম। তবে আমরা এ প্রোগ্রামটি নতুন ভাবে লিখব। রিকারশন ব্যবহার করে। প্রোগ্রামটি যেহেতু সিম্পল, তাহলে আমরা একটু ভালো করে দেখলেই বুঝতে পারব। আর প্রোগ্রামটি কিভাবে কাজ করে, তা বুঝতে পারলেই আমরা রিকার্শন বুঝে ফেলব। এরপর আমরা কমপ্লেক্স সব প্রোগ্রাম রিকারশন ব্যবহার করে সহজেই লিখে ফেলতে পারব। আগে ফাংশনটি **Pseudo code [সুডো কোড]** এ আগে লিখি, এরপর সি প্রোগ্রামিং এ ইমপ্লিমেন্টেশন দেখব।

[**Pseudo code**](https://en.wikipedia.org/wiki/Pseudocode) হচ্ছে একটা প্রোগ্রাম বা একটা অ্যালগরিদমের ধাপ গুলো সাধারণ মানুষের ব্যবহার উপযোগি করে লেখা কিছু কোড। এগুলো কোন প্রোগ্রামিং ল্যাঙ্গুয়েজ ব্যবহার করে লেখা হয় না। এমন ভাবে লেখা হয় যেন মানুষ বুঝতে পারে। নিচে printInt নামে একটি ফাংশনের সুডো কোড দেওয়া হলো, যা ১ থেকে ঐ সংখ্যা পর্যন্ত সকল সংখ্যা প্রিন্ট করবে।

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | printInt( k ) {      if (k == 0) {      return 0;  }  print(k );  printInt( k - 1 );  } |

ফাংশনটি প্যারামিটার হিসেবে একটা ইন্টিজার নিবে। এরপর চেক করবে সংখ্যাটা কি ০? যদি শুন্য হয়, ঐখানেই ফাংশনের কাজ শেষ হবে। যদি ০ না হয় তাহলে ইন্টিজারটি প্রিন্ট করবে। এবং ঐ ইন্টিজার সংখ্যাটি থেকে ১ বিয়োগ করে আবার printInt কে কল করবে। মানে নিজেকে নিজে কল করবে।