>>> 2 \*\* 3

8

>>> 3 \*\* 3

27

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

>>> 10 // 3

3

>>> 10 % 3

1

# স্ট্রিং

পাইথনে যে কোন সেনটেন্সকেই স্ট্রিং হিসেবে ব্যবহার করা যায় সিঙ্গেল(' '), ডাবল("") কিংবা ট্রিপল(""" """) কোটেশন এর মাধ্যমে।

>>> "We love python!"

'We love python!'

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

"Brian's mother: He's not the Messiah. He's a very naughty boy!"

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

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

>>> """Me: Hi, there!

... She: Yes, please!"""

'Me: Hi, there!\nShe: Yes, please!'

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

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

# ব্যাসিক ইনপুট আউটপুট

## **ইনপুট ফাংশন** input()

>>> input("Give me your country name: ")

Give me your country name: bangladesh

>>>bangladesh

'bangladesh'

এ ক্ষেত্রে পাইথন ইন্টারপ্রেটার যখন কোডটা এক্সিকিউট করবে তখন ইন্টারপ্রেটার ইউজার এর ইনপুট এর জন্য অপেক্ষা করবে এবং ততক্ষণ পর্যন্ত ইনপুট এর ডাটা গ্রহণ করবে যতক্ষণ না ইউজার Enter বাটন প্রেস করে অথবা ইনপুটে আরেকটি নিউলাইন ক্যারেক্টার আসে

## **আউটপুট ফাংশন** print()

>>> print('Hello World!')

Hello World!

>>> print(5+5)

10

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

Name: Bangladeash

Population: 156.6M

# স্ট্রিং অপারেশন

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

>>> "Spam" + 'eggs'

'Spameggs'

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

First string, second string

তাই বলে কোন নাম্বারের সাথে স্ট্রিং যোগ করা যাবে না,

1 + '2' + 3 + '4'

Traceback (most recent call last):

## **রিপিটেশন (Repetition)**

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

>>> print("spam" \* 3)

spamspamspam

>>> 4 \* '2'

'2222'

>>> '17' \* '87'

Traceback (most recent call last):

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

TypeError: can't multiply sequence by non-int of type 'str'

>>> 'pythonisfun' \* 7.0

Traceback (most recent call last):

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

TypeError: can't multiply sequence by non-int of type 'float'

## **স্ট্রিং ফরম্যাটিং**

নন স্ট্রিং ডাটার সাথে স্ট্রিং টাইপের ডাটাকে যুক্ত করে সুন্দর স্ট্রিং আউটপুট তৈরি করতে format মেথড ব্যবহার করা হয়। এর মাধ্যমে একটি স্ট্রিং এর মধ্যে থাকা কিছু আর্গুমেন্টকে রিপ্লেস বা সাবস্টিটিউট করা যায়। format মেথডের মধ্যের প্রত্যেকটি আর্গুমেন্ট দিয়ে এর সামনে থাকা স্ট্রিং এর মধ্যের প্লেস হোল্ডার গুলোকে রিপ্লেস করা হয়। প্লেস হোল্ডার গুলো {} এর সাথে ইনডেক্স বা নাম ব্যবহার করে ডিফাইন করা হয়।

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

>>> print(msg)

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

>>> '{2}, {1}, {0}'.format('a', 'b', 'c')

'c, b, a'

>>> '{0}{1}{0}'.format('abra', 'cad')

'abracadabra'

## Format

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

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

>>> print(message)

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

## replace

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

>>> print(", ".join(["apple", "orange", "pineapple"]))

apple, orange, pineapple

## startswith, endswith

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

>>> print("This is a sentence.".startswith("This"))

True

>>> print("This is a sentence.".endswith("sentence."))

True

## upper() and lower()

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

>>> print("This is a sentence.".upper())

THIS IS A SENTENCE.

>>> print("AN ALL CAPS SENTENCE".lower())

an all caps sentence

## split

মেথড হচ্ছে join মেথডের উল্টো। অর্থাৎ একটি বাক্যেকে নির্দিষ্ট কোন সেপারেটর এর সাপেক্ষে ভেঙ্গে একটি লিস্ট তৈরি করা যায় এই মেথডের মাধ্যমে।

>>> print("a, e, i, o, u".split(", "))

['a', 'e', 'i', 'o', 'u']

# টাইপ কনভার্সন

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

## **ইন্টেজার এ কনভার্সন**

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

>>> int("123")

123

>>> int(12.3)

12

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

>>> int("123a")

Traceback (most recent call last):

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

ValueError: invalid literal for int() with base 10: '123a'

## **ফ্লোট এ কনভার্সন**

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

>>> float("123.456")

123.456

>>> float(123)

123.0

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

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

>>> float(123)

123.0

>>> float("123.456")

123.456

>>> int(123.456)

123

>>> int("123.456")

Traceback (most recent call last):

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

ValueError: invalid literal for int() with base 10: '123.456'

## স্ট্রিং এ কনভার্সন

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

>>> str(123)

'123'

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

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

Float = 10.5 Integer = 50

# ভ্যারিয়েবল

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

একটি ভ্যারিয়েবলের মধ্যে কোন ভ্যালু জমা রাখার জন্য একটি সমান (=) চিহ্ন ব্যবহার করা হয়।

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

>>> x = 7

>>> print(x)

7

>>> print(x + 3)

10

## রি-অ্যাসাইনমেন্ট

>>> x = 10.5

>>> print(x)

10.5

>>> x = "Hello There"

>>> print(x)

Hello There

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

### নামকরণ এর নিয়ম

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

* ভ্যারিয়েবলের নাম অবশ্যই এক শব্দের হতে হবে। অর্থাৎ এরকম ভাবে ভ্যারিয়েবল লেখা যাবে নাঃ

my variable = 10

* প্রথম অক্ষর অবশ্যই একটি alphabetic letter(uppercase or lowercase) অথবা underscore ( \_ ) হতে হবে । যেমনঃ nafis, a, b, \_variable লেখা যাবে ভ্যারিয়েবল হিসেবে কিন্তু 1name, @nafis, 7a, %b এই ভাবে লেখা যাবে না । প্রথম অক্ষর ছাড়া পরে letter, underscore, number ব্যবহার করা যাবে। যেমনঃ variable1, my\_variable, not\_Very\_Good\_Name10

যদিও ভ্যারিয়েবলের শুরুতে underscore ব্যবহার করা যায়, কিন্তু পাইথনের কনভেনশন হচ্ছে ভ্যারিয়েবলের নাম সবসময় lowercase letter দিয়ে শুরু করা।

* পাইথন Case Sensitive অর্থাৎ a = 4 এবং A = 4 একই ভ্যারিয়েবল না।
* পাইথনের কিছু reserved কী-ওয়ার্ড আছে, এগুলো ব্যবহার করা যাবে না । যেমনঃ if, else, elif, for, while, break, continue, except, as, in, is, True, False, yield, None, def, del, class ইত্যাদি।
* >>> this\_is\_a\_normal\_name = 7
* >>> 123abc = 7
* SyntaxError: invalid syntax
* >>> spaces are not allowed
* SyntaxError: invalid syntax

## রিভিউ

নিচের উদাহরণে আমরা দেখবো কিভাবে সেই ফাংশন ব্যবহার করে এবং একটি ভ্যারিয়েবল ব্যবহার করে ইউজার এর ইনপুট মেমোরিতে জমা রাখা যায় এবং পরবর্তীতে ব্যবহার করা যায়,

>>> user\_input = input("Enter your birth year: ")

Enter your birth year: 1994

>>> age = 2019 - int(user\_input)

>>> print("You are " + str(age) + " years old!")

You are 25 years old!

# ইনপ্লেস অপারেটর

>>>a = 3

>>>a += 2

>>> print(a)

5

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

## কমন ইনপ্লেইস অপারেটর:

* +=
* -=
* \*=
* /=
* %=

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

>>> language = "Python"

>>> language += "3"

>>> print(language)

Python3

# কন্ট্রোল স্ট্রাকচার

>>>Ey program gulo kono IDE use kore korte hobe(like:pycharm)

## if স্টেটমেন্ট

if 10 > 5:

     print("10 greater than 5")

10 greater than 5

## নেস্টেড if

num = 12

if num > 5:

    print("Bigger than 5")

    if num <= 47:

        print("Between 6 and 47")

Bigger than 5

Between 6 and 47

## else স্টেটমেন্ট

num = 12

if num > 5:

    print("Bigger than 5")

else:

    print("small then 5")

Bigger than 5

## if else চেইন

num = 7

if num == 5:

    print("Number is 5")

else:

    if num == 11:

        print("Number is 11")

    else:

        if num == 7:

            print("Number is 7")

        else:

            print("Number isn't 5, 11 or 7")

Number is 7

মজার ব্যাপার হচ্ছে এরকম if else if এর চেইনকে একটু সংক্ষেপে elif দিয়েও লেখা যায়

num = 7

if num == 5:

    print("Number is 5")

elif num == 11:

    print("Number is 11")

elif num == 7:

    print("Number is 7")

else:

    print("Number isn't 5, 11 or 7")

Number is 7

## **টারনারি অপারেটর**

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

a = 100

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

print(b)

200

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

status  = 1

msg = "Logout" if status == 1 else "Login"

print(msg)

## **else এর আরও ব্যবহার**

শুধুমাত্র if এর সাথে ব্যবহার বাদেও else কে ব্যবহার করা যায় for এবং while লুপের সাথেও।

for i in range(10):

    print(i)

else:

    print("Done")

## বুলিয়ান লজিক

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

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

### and

a= 1 == 1 and 2 == 2

print(a)

true

a= 1 == 1 and 2 == 3

print(a)

false

### or

a= 1 == 1 or 2 == 3

print(a)

true

### not

a= not 1 == 1

print(a)

false

## অপারেটর প্রেসিডেন্স

>>> False == False or True

True

>>> False == (False or True)

False

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

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

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

## while লুপ

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

i = 1

while i <= 5:

   print(i)

   i = i + 1

print("I am mahmud")

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

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

### infinite লুপ

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

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

while 1 == 2:

   print("In the loop")

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

কিবোর্ডের ctrl+C চেপে প্রোগ্রামটির কার্যক্রম বন্ধ করা যাবে।

### Break

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

i = 0

while 1 == 1:

   print(i)

   i = i + 1

   if i >= 5:

      print("Breaking")

      break

print("Finished")

### continue

এটি আরেকটি মজার জিনিষ। ধরা যাক, একটি লুপের মধ্যে আমরা বেশ কিছু কাজ করার স্টেটমেন্ট লিখেছি এবং চাচ্ছি যে লুপ যতক্ষণ সত্য থাকে (ধরে নেই ১০০ বার) ততক্ষণ এর মধ্যেকার কাজ গুলো বার বার হোক। কিন্তু, এমনও তো হতে পারে যে, সেই ১০০ বারের মধ্যে বিশেষ কয়েকবার আমরা সেই পুরো কাজটা করতে চাই না কিন্তু নির্ধারিত ১০০ বারই লুপকে কাজ করাতে দিতে চাই; তাহলে কি করতে পারি? এর জন্যই আছে continue. একটি while লুপের মধ্যে যখনই continue এক্সিকিউট হবে তখনিই লুপের মধ্যে থাকা এর পরের কোড গুলো এক্সিকিউট হবে না এবং লুপের কন্ট্রোল একদম শুরুতে চলে যাবে,

i = 0

while True:

   i = i +1

   if i == 2:

      print("Skipping 2")

      continue

   if i == 5:

      print("Breaking")

      break

   print(i)

print("Finished")

# লিস্ট

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

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

words = ["Hello", "world", "!"]

print(words[0])

print(words[1])

print(words[2])

## ফাকা list

my\_list = []

print(my\_list)

## list হিসেবে string

str = "Hello world!"

print(str[6])

# লিস্ট অপারেশন

my\_numbers = [1, 2, 3, 5]

my\_numbers[3] = 4

print(my\_numbers)

[1, 2, 3, 4]

first\_list = [1, 2, 3]

print(first\_list + [4, 5, 6])

print(first\_list \* 3)

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

[1, 2, 3, 1, 2, 3, 1, 2, 3]

## লিস্টের মধ্যের এলিমেন্ট চেক

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

print("orange" in fruits)

print("rice" in fruits)

print("apple" in fruits)

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

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

print("orange" not in fruits)

print(not "rice" in fruits)

# লিস্ট ফাংশন

## Append

কোন একটি লিস্টের শেষে নতুন এলিমেন্ট যুক্ত করতে এই মেথড ব্যবহার করা যাবে

nums = [1, 2, 3]

nums.append(4)

print(nums)

[1, 2, 3, 4]

## Insert

words = ["A", "C"]

index = 1

words.insert(index, "B")

print(words)

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

লিস্টের কোন একটি নির্দিষ্ট পজিশনে বা ইনডেক্সে কোন এলিমেন্ট যুক্ত করতে চাইলে append এ কাজ হবে না (কারন এটা শেষে যুক্ত করে) বরং insert ব্যবহার করতে হবে। insert মেথডের দুটো প্যারামিটার - প্রথমটি হচ্ছে লিস্টের কোন পজিশনে নতুন এলিমেন্ট যুক্ত করতে চান আর দ্বিতীয় প্যারামিটারটি হচ্ছে যে এলিমেন্ট যুক্ত করতে চান সেটি নিজেই

## Index

letters = ['p', 'q', 'r', 's', 'p', 'u']

print(letters.index('r'))

print(letters.index('p'))

print(letters.index('z'))

2

0

ValueError: 'z' is not in list

## Count

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

le = ['p', 'q', 'r', 's', 'p','p','p','p','p''u']

print(le.count('p'))

5

## max(), min(), len()

একটি লিস্টের মধ্যে থাকা এলিমেন্ট গুলোর মধ্যে থেকে বড়টি দেখে নিতে max() ফাংশনের ব্যবহার করা যেতে পারে।soto dekte min. and total element length dekte len use hoy

nums = [1, 2, 4, 20, 50, 3, 4]

print(max(nums))

print(min(nums))

print(len(nums))

50

1

7

## List স্লাইস

একটি লিস্টকে ভাগ করে আরেকটি লিস্ট তৈরি করা

some\_marks = [2, 4, 6, 32, 60, 65, 69, 76, 80, 85, 90]

avg\_marks = some\_marks[4:8]

print(avg\_marks)

good\_marks = some\_marks[8:]

print(good\_marks)

poor\_marks = some\_marks[:4]

print(poor\_marks)

[60, 65, 69, 76]

[80, 85, 90]

[2, 4, 6, 32]

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

## ইনডেক্স জাম্প

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

print(numbers[2:9:3])

[3, 6, 9]

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

## নেগেটিভ ইনডেক্স স্লাইস

squares = [1, 2, 5, 0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 4, 6, 7, 8]

print(squares[3:-4])

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

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

## লিস্ট রিভার্স

values = [3, 4, 5, 6, 7, 8]

print(values[::-1])

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

# Prints the minimum value among all the elements of the list below

print(min([1, 2, 3, 4, 0, 2, 1]))

# Prints the maximum value among all the elements of the following list

print(max([1, 4, 9, 2, 5, 6, 8]))

# Print sum of all the elements of the following list

print(sum([1, 2, 3, 4, 5]))

# বাক্য থেকে Vowel দূর করা

vowels = 'aeiou'

sentence = 'i am awesome!'

filtered\_letters = []

for letter in sentence:

    if letter not in vowels:

        filtered\_letters.append(letter)

print(''.join(filtered\_letters))

m wsm!

## লিস্ট কম্প্রিহেনশন ব্যবহার করে

vowels = 'aeiou'

sentence = 'I am awesome'

filtered\_sentence = ''.join([letter for letter in sentence if letter not in vowels])

print(filtered\_sentence)

# রেঞ্জ

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

my\_numbers = list(range(10))

print(my\_numbers)

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

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

সবসময় যে ০ থেকেই রেঞ্জ এর এলিমেন্ট শুরু হয় সেটা নয়

my\_numbers = list(range(5,10))

print(my\_numbers)

[5, 6, 7, 8, 9]

আমরা চাইলে শুরু এবং শেষ নাম্বারের মধ্যে একটি ইন্টারভাল সেট করতে পারি যাতে করে এলিমেন্ট গুলো ক্রমিক না হয়ে বরং নিধারিত বিরতির হবে।

my\_numbers = list(range(5, 30, 3))

print(my\_numbers)

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

# বুলিয়ান

বুলিয়ান হলো এক প্রকারের ডাটাটাইপ যার মান সবসময় কোন কিছু সত্য অথবা মিথ্যা বুঝায়। সত্য ও মিথ্যাকে যথাক্রমে 1 ও 0 দ্বারা প্রকাশ করা হয়. পাইথনে এই Boolean টাইপটির দুটি ভ্যালু আছে True এবং False

## **বুলিয়ান এক্সপ্রেশন**

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

## বুলিয়ান অপারেটর

বুলিয়ান টাইপের তিনটি বেসিক অপারেটর আছে। এরা হলো AND , OR , NOT

x=2 == 3

print(x)

False

x="hello" == "hello"

print(x)

True

x=1 != 1

print(x)

false

x="eleven" != "seven"

print(x)

true

x=7 > 5

print(x)

true

x=9 >= 9.0

print(x)

true

# for লুপ

for লুপ দিয়ে খুব সহজেই যেকোনো সিকোয়েন্স টাইপ অবজেক্ট যেমন list, string ইত্যাদির মধ্যে iterate করা যায়

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

for fruit in fruits:

    print(fruit + " Juice!")

যখনই কোন iterable নিয়ে কাজ করার প্রয়োজন পরবে তখন for লুপ ব্যবহার করাই ভালো হয়।

for i in range(100):

    print(i)

range ফাংশন ব্যবহার করে একটি কাল্পনিক লিস্ট তৈরি করা হয়েছে

for i in range(5, 10):

      print(i)

5, 6, 7, 8, 9

ধরুন, ৫ থেকে শুরু করে ৯ পর্যন্ত

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

for i in range(5, 15, 3):

      print(i)

5 , 8 , 11 , 14

# start with 10

# end with 0

# step size -2

for i in range(10, 0, -2):

     print(i)

10

8

6

4

2

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

    print(letter)

string তো একরকম iterable

for i in range(20):

    if i == 5:

        continue

    if i > 9:

        break

    print(i)

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

while লুপের মত ফর লুপেও break, continue ইত্যাদি কিওয়ার্ড ব্যবহার করে

# None

কিন্তু যদি এমন কোন ভ্যারিয়েবল নেই যার আসলে কোন ভ্যালুই নাই সেটা কিভাবে ইনিসিয়ালাইজ করা যেতে পারে? None হচ্ছে NoneType এর একটি অবজেক্ট যা দিয়ে আসলে ভ্যালুর অনুপস্থিতি নির্ধারণ করে দেয়া যায়। অন্যান্য ডাটা টাইপের যেমন একাধিক ভ্যালু থাকে পারে যেমন - bool টাইপের দুটো ভ্যালু হতে পারে; True অথবা False. NoneType এর একটাই ভ্যালু আর সেটা হল এই None.

>>> type(None)

<class 'NoneType'>

যখন কোন ফাংশন নির্দিষ্ট করে কোন কিছু রিটার্ন করে না তখন বস্তুত সে None রিটার্ন করে

def my\_func():

    print("Printing Hello")

what\_i\_got = my\_func()

print(what\_i\_got)

Printing Hello

None

# VIP SECTIONS STAR

# ডিকশনারি

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।

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

print(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

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

print(marks["Math"])

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 -

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

print(marks["Math"])

[60, 74, 58]

কিন্তু নিচের মত একটি ডিকশনারি হতে পারে না –

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

print(marks["Math"])

## ডিকশনারি ফাংশন

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

লিস্টের ক্ষেত্রে উদাহরণ

my\_list = [2, 4, 6, 7]

my\_list[3] = 8

print(my\_list)

8

ডিকশনারির ক্ষেত্রে –

my\_list = {1 : 1, 2 : 4, 3 : 9, 4 : "What?"}

my\_list[4] = 16

print(my\_list)

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

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

my\_list = [2, 4, 6, 8]

my\_list[4] = 10

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

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

my\_nums = {1 : 1, 2 : 4, 3 : 9, 4 : 16}

my\_nums[5] = 25

print(my\_nums)

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

অর্থাৎ নতুন key 5 এবং এর ভ্যালু 25 দুটোই my\_nums ডিকশনারিতে ম্যানুয়ালি যুক্ত করা হয়েছে।

## key খোঁজা

যদি আমরা চেক করতে চাই যে একটি ডিকশনারিতে নির্দিষ্ট কোন একটি key আছে কিনা তার জন্য in এবং not in ব্যবহার করা যায়।

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

print(1 in nums)

print("three" in nums)

print(4 not in nums)

True

False

True

## get এর ব্যবহার

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

my\_nums = {1 : 1, 2 : 4, 3 : 9, 4 : 16}

print(my\_nums[5])

Traceback (most recent call last):

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

KeyError: 5

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

my\_nums = {1 : 1, 2 : 4, 3 : 9, 4 : 16}

print(my\_nums.get(5))

None

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

my\_nums = {1 : 1, 2 : 4, 3 : 9, 4 : 16}

print(my\_nums.get(5, "5 not in my numbers!"))

## টাপল

লিস্টের মতই আরেকটি ডাটা স্ট্রাকচার হচ্ছে Tuple. কিন্তু গুরুত্বপূর্ণ পার্থক্যটি হচ্ছে এটি Immutable টাইপের অর্থাৎ, এর ভ্যালু পরিবর্তন করা যায় না। আবার, লিস্ট যেমন তৈরি করতে হয় দুটো [] ব্র্যাকেট দিয়ে কিন্তু টাপল তৈরি করতে হয় () দিয়ে (যদিও ব্র্যাকেট ছাড়াও শুধু কমা চিহ্ন দিয়ে ভ্যালু গুলোকে আলাদা করেও টাপল তৈরি করা যায়).

mah = ("Admin", "Operator", "User")

print(mah[0])

print(mah[2])

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

mah = ("Admin", "Operator", "User")

mah[2] = "Customer"

print(mah[0])

print(mah[2])

TypeError: 'tuple' object does not support item assignment

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

permissions = (("Admin", "Operator", "Customer"), ("Developer", "Tester"), [1, 2, 3], {"Stage": "Development"})

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

Development

## টাপল আনপ্যাকিং

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

numbers = (1, 2, 3)

a, b, c = numbers

print(a)

print(b)

print(c)

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

a, b, \*c, d = [1, 2, 3, 4, 5, 6, 7, 8, 9]

print(a)

print(b)

print(c)

print(d)

1

2

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

9

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

# ফাংশন ও মডিউল

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

## ফাংশন

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

# Defining the function named hello

def hello():

    print("Hello World!")

# Calling the function to use it

hello()

hello world

আমরা def কি-ওয়ার্ডটি লিখেছি । তারপর ফাংশনের একটা নাম দিয়েছি – hello, এবং তারপর একজোড়া প্রথম বন্ধনী (). এরপর একটি কোলন তথা : চিহ্ন দিয়ে এর নিচে ফাংশনের আওতাভুক্ত কোড ব্লক বা ফাংশনের কাজ ডিফাইন (নির্ধারণ) করেছি।আমরা print ফাংশন ব্যবহার করি যখন দরকার হয় তখন। উপরের ফাংশনটিকে কল করতে হবে নিচের মত করে, একটা ফাংশনকে প্রোগ্রামে একবার ডিফাইন করলেও সেটাকে বার বার কল বা ব্যবহার করা যাবে।

## ফাংশন আর্গুমেন্ট

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**।

def show\_double(x):

    print(x\*2)

show\_double(2)

show\_double(100)

4

200

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

আর্গুমেন্টকে ফাংশনের দুটি প্রথম বন্ধনীর মধ্যে ডিফাইন করতে হয়।

একটি ফাংশন কিন্তু একাধিক আর্গুমেন্ট নিয়ে কাজ করতে পারে অর্থাৎ এর একাধিক আর্গুমেন্ট থাকতে পারে।

def make\_sum(x, y):

    z = x + y

    print(z)

make\_sum(5, 10)

make\_sum(500, 500)

15

1000

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

কিন্তু এই x, y বা z কে উক্ত ফাংশনের বাইরে থেকে অ্যাক্সেস করা যাবে না বা ব্যবহার করা যাবে না। যেমন –

def make\_sum(x, y):

    z = x + y

    print(z)

make\_sum(5, 10)

print(z)

15

...

NameError: name 'z' is not defined

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

## মাল্টিপল প্যারামিটার হ্যান্ডলিং | আর্বিটরারি আর্গুমেন্ট লিস্ট

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

def make\_sum(\*args):

    sum = 0

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

        sum += num

    return sum

print(make\_sum(10, 20, 30, 40))

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. 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।

def print\_dict(\*args):

    print (args)

print\_dict(a=1, b=2)

TypeError Traceback (most recent call last)

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

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

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

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

def print\_dict(\*\*kwargs):

    print(kwargs)

print\_dict(a=1, b=2, c=3)

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

আমরা যদি কোডটা আরেকটু গুছিয়ে লেখি,

def print\_dict(\*\*kwargs):

    for args in kwargs:

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

print\_dict(a=1, b=2, c=3)

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

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

    print(a)

    print(args)

    print(kwargs)

print\_all(10, 20, 30, 50, b=5, c=10)

## 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।

# ফাংশন রিটার্ন

কোন একটি ফাংশন তার বডির মধ্যে কিছু কাজ করে চুপ চাপ থাকতে পারে অথবা কোন কিছু রিটার্নও দিতে পারে। কাকে রিটার্ন দিবে? যে এই ফাংশনকে কল করবে বা ব্যবহার করবে। কোথায় রিটার্ন দিবে? যেখান থেকে কল করা হবে সেখানেই রিটার্ন ভ্যালু পৌঁছে যাবে।

def get\_larger(x, y):

    if x > y:

        return x

    else:

        return y

larger\_value = get\_larger(23, 32)

print(larger\_value)

32

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

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

def add\_numbers(x, y):

    total = x + y

    return total

    print("This won't be printed")

print(add\_numbers(4, 5))

9

উপরের প্রোগ্রামের ফাংশনটির কাজ হচ্ছে দুটো আর্গুমেন্ট ভ্যারিয়েবলকে যোগ করে নতুন একটি ভ্যারিয়েবলে রেখে সেটিকে তার caller এর কাছে রিটার্ন করা । রিটার্ন করেই সে ক্ষান্ত। এরপরে আরেকটি প্রিন্ট স্টেটমেন্ট থাকলেও সেটার কোন এক্সিকিউশন নেই।

# কমেন্ট ও ডক স্ট্রিং

## কমেন্ট/single comment

Syntax: #

পাইথনে যেকোনো কমেন্ট লাইন লেখার আগে তার আগে # (হ্যাস বা পাউন্ড) চিহ্ন ব্যবহার করতে হয়

# few variables below

x = 10

y = 5

# make sum of the above two variables

# and store the result in z

z = x + y

print(z) # print the result

# print (x // y)

# another comment

## ডক স্ট্রিং/multiple comment

Syntax: must be three “””” add

”””

Hi mamm

Hhhdhdhdh

Dhdhhdh

“””

def greet(word):

    """"

    Print a word with an

    exclamation mark following it.

    """

    print(word + "!")

greet("Hello World")

\*\*\*তবে পাইথনের মাল্টি লাইনের কমেন্ট কে print(\_\_doc\_\_) এই function দিয়ে print করে দেখা যায়।

"""

Here's a long Python comments spread over

many lines.

You can format a multi-line comment

any way you like.

"""

print(\_\_doc\_\_)

# অবজেক্ট হিসেবে ফাংশন

ফাংশনও একটি অবজেক্ট অর্থাৎ এরও কিছু অ্যাট্রিবিউট ও মেথড আছে. সাধারণ ভ্যারিয়েবলের ভ্যালুর মত কোন একটি ফাংশনকেও একটি ভ্যারিয়েবলে অ্যাসাইন বা স্টোর করা যায়।

def add\_explanation(line):

    return line + '!'

update\_line = add\_explanation

print(update\_line("Hello World"))

Hello World!

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

যেহেতু ফাংশনকে ভ্যারিয়েবলে অ্যাসাইন করা যায় তাহলে কি ভ্যারিয়েবলের মত করে একটা ফাংশনকেও অন্য ফাংশনের আর্গুমেন্ট হিসেবে পাঠানো যাবে

def beautify(text):

    return text + '!!!'

def make\_line(func, words):

    return "Hello " + func(words)

print(make\_line(beautify, "world"))

Hello world!!!

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

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

# মডিউল

মডিউল হচ্ছে কিছু কোডের সমষ্টি যেখানে বেশ কিছু ফাংশন, ভ্যারিয়েবল বা ডাটা থাকে এবং যেগুলোকে অ্যাক্সেস করে প্রয়োজনে আরেকটি পাইথন প্রোগ্রামে ব্যবহার করা যায়। পাইথনের অনেক অনেক বিল্ট-ইন মডিউল আছে. নতুন একটি প্রোগ্রামে এরকম কোন মডিউল ব্যবহার করতে চাইলে প্রথমেই সেটিকে import করে নিতে হবে। import MODULE\_NAME এভাবে। এবার এই স্টেটমেন্টের নিচে MODULE\_NAME.VAR এভাবে উক্ত মডিউলের ফাংশন বা ভ্যারিয়েবলকে অ্যাক্সেস করা যাবে

import random

value = random.randint(1, 100)

print(value)

উপরের প্রোগ্রামে value নামের ভ্যারিয়েবলে আমরা একটি র‍্যান্ডম নাম্বার ষ্টোর করতে চেয়েছি। যে র‍্যান্ডম নাম্বারটি হবে ১ থেকে ১০০ এর মধ্যে। কিন্তু আমরা নিজেরা সেই র‍্যান্ডম নাম্বার তৈরির ফাংশন লিখি নাই। বরং আমরা পাইথনের একটি বিল্ট ইন মডিউল random কে ইম্পোর্ট করে নিয়েছি এবং এর মধ্যে আগেই ডিফাইন করে রাখা randint ফাংশনকে ব্যবহার করে র‍্যান্ডম নাম্বার পাচ্ছি।

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

from math import pi, sqrt

print(pi)

print(sqrt(25))

উপরের উদাহরণে আমরা math মডিউল থেকে শুধুমাত্র pi কন্সট্যান্টটি এবং sqrt ফাংশনটিকে ইম্পোর্ট করেছি। আপনি চাচ্ছেন square root বের করার ফাংশনের নাম আরেকটু সুন্দর হলে ভালো হয়।

from math import sqrt as square\_root

print(square\_root(25))

5

## স্ট্যান্ডার্ড লাইব্রেরী

<https://python.howtocode.com.bd/function-and-module/standard-library>

## pip

অন্যদের ডেভেলপ করা মডিউল গুলোকে পাওয়া যায় [PyPI - the Python Package Index](https://pypi.python.org/pypi) এখানে।

For install any model package : pip install LIBRARY\_NAME

# এক্সেপশন

যখন একটি প্রোগ্রামের স্বাভাবিক এক্সিকিউশনের মধ্যে কোন বাধার উৎপত্তি হয়। অর্থাৎ যখন একটি পাইথন স্ক্রিপ্ট এমন কোন একটি সমস্যাপূর্ণ অবস্থার সম্মুখীন হয় যা সে এড়িয়ে যেতে পারে না অথবা সমাধান করতে পারে না অতঃপর প্রোগ্রামের এক্সিকিউশন বন্ধ হয়ে যায় - সেরকম ঘটনাকে এক্সেপশন বলা. সাধারণত ভুল কোড বা ইনপুটের জন্য প্রোগ্রামের মধ্যে এক্সেপশন তৈরি হয় যা সঠিকভাবে হ্যান্ডেল না করলে প্রোগ্রাম অনাকাঙ্ক্ষিত ভাবে বন্ধ হয়ে যেতে পারে.

a = 2500

b = 0

print(a/b)

print("I did it")

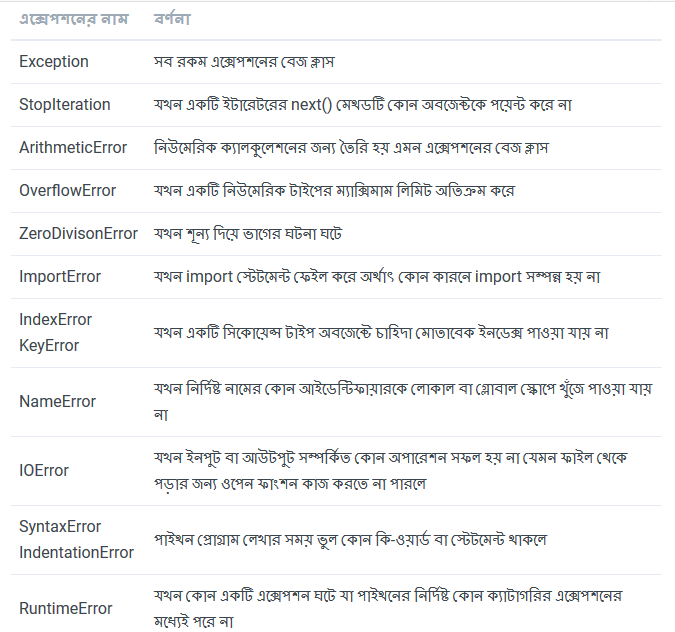
Traceback (most recent call last):

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

ZeroDivisionError: division by zero

উপরের প্রোগ্রামে, গণিতের নিয়ম অনুযায়ী a কে b দিয়ে ভাগ করা সম্ভব না আর তাই যখনই print(a/b) স্টেটমেন্টটি এক্সিকিউট হতে চেয়েছে তখনি এক ধরণের ব্যতিক্রম অবস্থার উৎপত্তি হয়েছে যাকে এক্সেপশন বলা হচ্ছে।

আর তাই পাইথন ওই প্রোগ্রামের পরবর্তী স্টেটমেন্ট গুলো এক্সিকিউট না করে বরং প্রোগ্রাম এক্সিকিউশন বন্ধ করে দিয়েছে। কিন্তু দেখা যাচ্ছে কোডের লেখায় বা নিয়মে কিন্তু কোন ভুল নাই। শুধুমাত্র রান টাইমেই এই পরিস্থিতি তৈরি হয়েছে। তাই এরকম অবস্থায় পাইথন এক্সেপশন তৈরি করে।



# এক্সেপশন হ্যান্ডেলিং

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

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

try:

    a = 1000

    b = int(input("Enter a divisor to divide 1000: "))

    print(a/b)

except ZeroDivisionError:

    print("You entered 0 which is not permitted!")

যদি ইনপুট হয় নিচের মত,

Enter a divisor to divide 1000: 5

তাহলে আউটপুট,

200.0

অথবা যদি ইনপুট হয় এরকম,

Enter a divisor to divide 1000: 0

তবে আউটপুট,

You entered 0 which is not permitted!

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

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

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

try:

    variable = 10

    print(variable + "hello")

    print(variable / 2)

except ZeroDivisionError:

    print("Divided by zero")

except (ValueError, TypeError):

    print("Type or value error occurred")

Type or value error occurred

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

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

try:

    word = "spam"

    print(word / 0)

except:

    print("An error occurred")

An error occurred

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

# Finally

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

try:

   print("Hello")

   print(1 / 0)

except ZeroDivisionError:

   print("Divided by zero")

finally:

   print("This code will run no matter what")

Hello

Divided by zero

This code will run no matter what

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

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

try:

   print(1)

   print(10 / 0)

except ZeroDivisionError:

   print(unknown\_var)

finally:

   print("This is executed last")

Traceback (most recent call last):

File "c:/Users/Mahmud Hossain/Desktop/folder1/runn.py", line 5, in <module>

print(unknown\_var)

NameError: name 'unknown\_var' is not defined

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

# এক্সেপশন Raise

raise স্টেটমেন্ট ব্যবহার করে এভাবে কাস্টম এক্সেপশন তৈরি করা যায়.

print("Hello")

raise NameError('HiThere')

Hello

Traceback (most recent call last):

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

NameError: HiThere

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

raise এর একটি মজার ব্যবহার দেখবো নিচের উদাহরণে

try:

    num = 5 / 0

except:

    print("Custom message about an error!")

    raise

Custom message about an error!

Traceback (most recent call last):

File "/Users/nuhil/Desktop/Test.py", line 2, in <module>

num = 5 / 0

ZeroDivisionError: integer division or modulo by zero

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

# Assertions

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

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

print(1)

assert 2 + 2 == 4

print(2)

assert 1 + 1 == 3

print(3)

1

2

Traceback (most recent call last):

File "/Users/nuhil/Desktop/Test.py", line 4, in <module>

assert 1 + 1 == 3

AssertionError

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

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

# ফাইল খোলা/ফাইল পড়া/ফাইলে লেখা/

## Open a exacting file

file\_on = open("mah.txt")

open ফাংশনের আর্গুমেন্ট হিসেবে আলোচ্য ফাইলের পাথ দিতে হয়। যদি পাইথন স্ক্রিপ্ট এবং ফাইলটি কম্পিউটারের একই লোকেশনে থাকে তাহলে উপরের মত শুধু ফাইলের নামটি লিখলেই কাজ শেষ। না হলে

file\_to\_work\_on = open("/Users/nuhil/Desktop/file\_name.txt") এভাবে লিখতে হতে পারে।

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

## Create /write a text file

file\_to\_work\_on = open("file\_name.txt", "w")

Example:

file\_to\_work = open("Test.txt", "w")

file\_to\_work.write("I am writing!!!")

file\_to\_work.close()

file\_to\_work = open("Test.txt", "r")

print(file\_to\_work.read())

file\_to\_work.close()

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

## Read a file

file\_to\_work\_on = open("file\_name.txt", "r")

Example:

file\_to\_work = open("Test.txt", "r")

content = file\_to\_work.read()

print(content)

file\_to\_work.close()

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

এভাবে পুরো কন্টেন্ট একসাথে না পড়ে বাইট হিসেবেও পড়া যায়। read মেথডের আর্গুমেন্ট হিসেবে কত বাইট পড়তে চাই সেটা পাঠিয়ে দেয়া যায়

file\_to\_work = open("Test.txt", "r")

just\_one\_character = file\_to\_work.read(1)

print(just\_one\_character)

remaining\_four\_characters = file\_to\_work.read(4)

print(remaining\_four\_characters)

rest\_of\_the\_file = file\_to\_work.read()

print(rest\_of\_the\_file)

file\_to\_work.close()

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

এবার লাইন বাই লাইন নিয়ে কাজ করার জন্য ফর লুপ ব্যবহার

file\_to\_work = open("Test.txt", "r")

for my\_line in file\_to\_work:

    print(my\_line)

file\_to\_work.close()

দেখুন কিভাবে আলাদা করে read বা readlines মেথড ব্যবহার না করেই সরাসরি ফর লুপ ব্যবহার করে প্রত্যেকটি লাইনকে অ্যাক্সেস করা যায়

## Append a file

file\_to\_work\_on = open("file\_name.txt", "a")

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

file\_to\_work\_on = open("my\_file", "wb")

ফাইল খোলার পর সেটি নিয়ে কাজ শেষে গুরুত্বপূর্ণ আরেকটি টাস্ক হচ্ছে সেই ফাইলকে ক্লোজ বা বন্ধ করা। নাহলে অকারণেই পাইথনের কাছে ফাইলটি ওপেন অবস্থাতেই থাকবে যা বস্তুত মেমোরি দখল করে থাকবে এবং প্রোগ্রামের পারফর্মেন্সে খারাপ ভূমিকা রাখবে।

file\_to\_work = open("filename.txt", "w")

# do HERE whatever you like, with the file

# such as write new lines in it

# then close it

file\_to\_work.close()

# ফাইল নিয়ে সঠিক কাজ

try:

    file\_to\_work = open("Test.txt", "r")

    content = file\_to\_work.read()

    print(content)

finally:

    file\_to\_work.close()

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

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

with open("Test.txt") as f:

    print(f.read())

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

# ল্যামডা

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

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।

def my\_function(func, arg):

    return func(arg)

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

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)।

print((lambda x,y: x + 2 \* y)(2,3))

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।

# ম্যাপ ও ফিল্টার

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

## ম্যাপ

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

def make\_double(x):

    return x \* 2

my\_marks = [10, 12, 20, 30]

result = map(make\_double, my\_marks)

print(list(result))

[20, 24, 40, 60]

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

## ফিল্টার

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

ফিল্টার সেই সমস্থ এলিমেন্ট কেই রিমুভ করে যার জন্য এর কাছে পাঠানো ফাংশনের রিটার্ন তথা প্রেডিকেট মিথ্যা হয়।

def is\_even(x):

    return x % 2 == 0

my\_numbers = [1,2,3,4,5,6]

result = filter(is\_even, my\_numbers)

print(list(result))

[2, 4, 6]

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

Main structure/syntax:

def my\_decorator(func):

        print("-----\*\*\*\*\*---------")

Main structure part

        func()

        print("-----\*\*\*\*\*---------")

Showing text

Showing part

def mahmud():

    print("I am Mahmud")

Calling text

decorated\_function = my\_decorator(mahmud)

if we want to add more style text, just define and call.

def my\_decorator(func):

   # def decorate():

        print("-----\*\*\*\*\*---------")

        func()

        print("-----\*\*\*\*\*---------")

   # return decorate

def mahmud():

    print("I am Mahmud")

decorated\_function = my\_decorator(mahmud)

#decorated\_function()

def hossain():

    print("I am Hossain")

decorated\_function = my\_decorator(hossain)

def lamyaa():

    print("I am  Lamyaa")

decorated\_function = my\_decorator(lamyaa)

-----\*\*\*\*\*---------

I am Mahmud

-----\*\*\*\*\*---------

-----\*\*\*\*\*---------

I am Hossain

-----\*\*\*\*\*---------

-----\*\*\*\*\*---------

I am Lamyaa

-----\*\*\*\*\*---------

# রিকারসন

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

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

বাস্তব উদাহরণ:

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

এটাকে এভাবেও চিন্তা করা যায়,

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

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

def factorial(x):

  if x == 1:

    return 1

  else:

    return x \* factorial(x-1)

print(factorial(5))

120

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

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

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

def is\_even(x):

  if x == 0:

    return True

  else:

    return is\_odd(x-1)

def is\_odd(x):

  return not is\_even(x)

print(is\_odd(17))

print(is\_even(23))

True

False

# সেট

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

num\_set = {1, 2, 3, 4, 5}

word\_set = set(["spam", "eggs", "sausage"])

print(3 in num\_set)

print(33 in num\_set)

print("spam" not in word\_set)

True

False

False

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

**সেটের কিছু গুরুত্বপূর্ণ বৈশিষ্ট্য**

* সেটের এলিমেন্ট গুলোর কোন ক্রম নেই অর্থাৎ এদেরকে ইন্ডেক্সিং করা যায় না
* একটি সেটে একই এলিমেন্ট একাধিক বার থাকতে পারে না
* একটি এলিমেন্ট কোন একটি সেটের অংশ কিনা সেটা খুব দ্রুত চেক করা যায়, লিস্ট এর তুলনায়
* # Has some duplicate eliments such as 1
* nums = {1, 2, 1, 3, 1, 4, 5, 6}
* print(nums)
* # To add an eliment to the set
* nums.add(-7)
* # To remove an element to the set
* nums.remove(3)
* print(nums)
* {1, 2, 3, 4, 5, 6}
* {1, 2, 4, 5, 6, -7}

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

## গণিতের সাথে তুলনীয় কিছু অপারেশন

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

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

first = {1, 2, 3, 4, 5, 6}

second = {4, 5, 6, 7, 8, 9}

print(first | second)

print(first & second)

print(first - second)

print(second - first)

print(first ^ second)

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

{4, 5, 6}

{1, 2, 3}

{8, 9, 7}

{1, 2, 3, 7, 8, 9}

## কিছু সিদ্ধান্ত

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

নিচের অনুসিদ্ধান্ত গুলো কাজে আসতে পারে,

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

# Itertools

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

from itertools import count

for i in count(3):

    print(i)

    if i >= 11:

        break

# Python এ statement কি?

পাইথন ইন্টারপ্রেটার(interpreter) যে ইন্সট্রাকশনসমূহ(Instructions) একটি Complete Code হিসেবে সম্পাদন করে তাকেই statement বলে।

Python এ statement দুই ধরণের :

১. Single Line Statement  
২. Multi Line Statement

## Single Line Statement

যখন একটি কমপ্লিট কোড একটি লাইনের মধ্যে লেখা হয় তখন এটাকে বলা হয় Single Line Statement. যেমন:

name="Badhon";

sum=5+8

print (name)

print (sum)

## Multi Line Statement

যখন একটি কমপ্লিট কোড একাধিক লাইনের মধ্যে লেখা হয় তখন এটাকে বলা হয় Multi Line Statement. পাইথনে আপনি যখন নতুন line এ যাবেন তখন স্বয়ংক্রিয়ভাবেই পূর্বের স্টেটমেন্ট শেষ হয়ে যায়। কিন্তু আপনি চাইলে লাইন কন্টিনিউয়েশন ক্যারেক্টার (\) দ্বারা মাল্টিপল লাইনের স্টেটমেন্ট তৈরি করতে পারবেন। যেমন:

total = 5 + 2 + 3 +\

        4 + 3 + 6 +\

        5 + 8 + 9

Print(‘total’)

পাইথনে Parenthesis( ), ব্রাকেট[ ] এবং ব্রাসেস{ } ব্যবহার করলে স্বয়ংক্রিয়ভাবেই লাইন চলমান থাকে। এক্ষেত্রে লাইন কন্টিনিউয়েশন ক্যারেক্টার ব্যবহার করতে হয় না। যেমন:

students= ['Iqbal',

          'Iftekhar',

          'Jony',

          'Ridoy']

total = (6 + 2 + 3 +

4 + 8 + 6 +

2 + 8 + 9)

print (total)

# Python এ Identifiers কি?

Python এ Identifier হচ্ছে একটি নাম যা variable, function, class, module এবং অন্যান্য object গুলোকে আইডেন্টিফাই বা চেনার জন্য ব্যবহার করা হয়। Python এ Identifier ঘোষণা করতে হলে যেকোনো ইংরেজি বর্ণ (লেটার ) (a-z, A-Z) অথবা আন্ডারস্কোর ( \_\_ )দিয়ে শুরু করতে হয়। Python এ Identifier এর নাম case sensitive.যেমন Python এর দৃষ্টিতে, a এবং A দুটি আলাদা Identifier. Identifier নামে কোনো ফাঁকা (স্পেস) থাকা যাবেনা।যদি নাম একের অধিক হয় তাহলে “\_”underscore (first\_name) অথবা বড় হাতের অক্ষরে(firstName) লিখতে হবে। আইডেন্টিফায়ার যেকোনো দৈর্ঘ্যের হতে পারে, তবে keyword গুলোকে আইডেন্টিফায়ার হিসাবে ব্যবহার করা যাবে না। Python Identifier গুলোর মধ্যে special symbols গুলো যেমন @ $ এবং % ইত্যাদি allow বা অনুমোদন করেনা।

## Python Identifiers গুলোর আরো কিছু naming Convention দেওয়া হলো :

* Python এ class name Identifier কে uppercase Later দিয়ে শুরু করতে হয়, অন্য সব Identifier গুলোকে lowercase later দিয়ে শুরু করতে হয়।

Expel:

Class Mahmud():

 যখন একটি Identifier একটি (single) (\_) underscore দিয়ে শুরু হয়,তাহলে এটিকে private identifier হিসেবে ধরা হয়।

 যখন একটি Identifier দুইটি (double) (\_\_) underscore দিয়ে শুরু হয়,তাহলে এটিকে strongly private identifier হিসেবে ধরা হয়।

 যখন একটি Identifier দুইটি (double) (\_\_) underscore দিয়ে শেষ হয়,তাহলে এটিকে language-defined special name. হিসেবে ধরা হয়।

# Python এ Keywords কি?

Python অথবা যেকোনো Programming Language এ keywords হচ্ছে সংরক্ষিত শব্দ(word) যা সিনট্যাক্স এর অংশ। আপনি নিজের কাজের জন্য এগুলো ব্যবহার করতে পারবেননা। অর্থাৎ variable, constant, function, class অথবা অন্য কোনো আইডেন্টিফায়ার এর নাম হিসেবে এগুলো ব্যবহার করতে পারবেন না। Python এর keywords সমূহ case-sensitive.True, False এবং None কীওয়ার্ড ছাড়া অন্য সকল কিওয়ার্ডসমূহ ছোট হাতের অক্ষর(lowercase) লিখতে হয় এবং লিখার সময় হুবুহু লিখতে হবে

|  |  |  |
| --- | --- | --- |
| and | exec | not |
| assert | finally | or |
| break | for | pass |
| class | from | print |
| continue | global | Raise |
| def | if | Return |
| del | import | Try |
| elif | in | While |
| else | is | With |
| except | lambda | Yield |

# Python এ variable কি

Python এ Variable হচ্ছে কম্পিউটারের মেমোরিতে অস্থায়ী তথ্য সংরক্ষণের পাত্র বা পাত্রের নাম , যা আমরা কম্পিউটারের মেমোরিতে যেকোনো অস্থায়ী তথ্য সংরক্ষণের জন্য ব্যবহার করে থাকি। কোন একটা variable এ একবার তথ্য রেখে সেটা পুরো কোডজুরে বারবার ব্যাবহার করতে পারেন, মুল তথ্য(value) টি বারবার রাখার পরিবর্তে। Python এ variable ঘোষণা করতে হলে যেকোনো ইংরেজি বর্ণ (লেটার ) (a-z, A-Z) অথবা আন্ডারস্কোর ( \_\_ )দিয়ে শুরু করতে হয়। Python এ variable এর নাম case sensitive.যেমন Python এর দৃষ্টিতে, a এবং A দুটি আলাদা ভেরিয়েবল। Variable নামে কোনো ফাঁকা (স্পেস) থাকা যাবেনা।যদি নাম একের অধিক হয় তাহলে “\_\_\_”underscore (first\_name) অথবা বড় হাতের অক্ষরে(firstName) লিখতে হবে। একটা variable এর মান স্ক্রিপ্টে যেকোন সময় পরিবর্তন হতে পারে। Python এ আপনাকে variable এর type ও ঘোষণা করতে হয় না। আমারা যখন কোনো variable এর value assign করি তখন এটি value এর উপর ভিত্তিকরে নিজ থেকেই type নির্ধারণ করে।

## Variable এ value assignment

Variable এ value assign করার জন্য Python এ আপনাকে assignment Operator (=) ব্যবহার করতে হবে । চলুন নিচের উদাহরণ গুলোর দিকে লক্ষ্য করা যাক :

a= 24

c= 23.6

str= "Habib"

print(a)

print (c)

print(str)

## একটি single statement এ Multiple Variable Assignment

Python Language এ আপনি অনেক গুলো Variable Assignment একই statement এ রাখতে পারবেন। যেমন-

a, c, str= 15, 15.4, "Araf"

print(a)

print (c)

print(str)

Python Language এ আপনি চাইলে একাধিক variable এ একই value assign করতে পারেন। নিচের উদাহরণটি অনুসরণ করুন:

a = b = c = "w3programmers"

print(b)

# পাইথনে ফাঙ্কশন কি? What is function in Python?

Python এ function হচ্ছে Program এর এমন একটা code block যা একটা নির্দিষ্ট কার্য সম্পাদনের এবং তা বার বার ব্যবহারের জন্যে ব্যবহৃত হয়। অর্থাৎ function এর মধ্যে এমন কিছু Instruction বা নির্দেশাবলী থাকে যা function ব্যবহারকারী থেকে এক বা একাধিক input নিয়ে এক বা একাধিক output বা result রিটার্ন করার জন্য ব্যবহৃত হয়। তবে অনেক সময় ইনপুট ছাড়াও আমরা আউটপুট রিটার্ন করতে পারি।

ফাঙ্কশনের মাধ্যমে প্রোগ্রামের যে অংশগুলো বার বার আসে সেগুলোকে আমরা পুনরায় ব্যবহারযোগ্য করতে পারি ।

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

পাইথনে সাধারণত, program page লোড হওয়ার সাথে সাথেই ডিক্লেয়ার করা function execute হয়না, যতক্ষণ না function কে call করা না হয়।

## পাইথনে function কত প্রকার ?

দুই প্রকার :

* Python Pre-built function বা Python language কর্তৃক পূর্ব নির্ধারিত বিভিন্ন library function বা library ছাড়া function
* User/Programmer defined function বা Programmer কর্তৃক নির্ধারিত function

## পাইথনে function কিভাবে লিখতে হয়?

* পাইথনে ফাংশন ডিক্লেয়ার করার জন্য def কি-ওয়ার্ডটি দিয়ে শুরু করতে হবে।
* def কি-ওয়ার্ডটির পর ফাঙ্কশনের নামটি letter অথবা underscore দিয়ে শুরু করতে হয়।
* ফাঙ্কশনের নামের শেষে একজোড়া প্রথম বন্ধনী () এরপর একটি colon (:) দিতে হবে।
* ফাংশনকে কল করার বা ব্যবহার করার আগেই সেই ফাংশনকে প্রোগ্রামে ডিফাইন করতে হবে।
* “””docstring””” অর্থাৎ function এর মধ্যে function এর documentation লেখার জন্য ব্যবহৃত হয়।
* পাইথনে function Case sensitive অর্থাৎ একই নামে কোনো ফাঙ্কশন lowercase এবং uppercase এর কারণে আলাদা function হিসেবে বিবেচিত হবে।

## পাইথনে function লেখার Syntax কি ?

|  |  |
| --- | --- |
| 1  2  3 | def function\_name(parameters):      """docstring"""      statement(s) |

চলুন syntax অনুসারে একটা function তৈরী করে ফেলি :

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | def greet(name):      """This function greets to      the person passed in as      parameter"""      print("Hello, " + name + ". Good morning!");    def Greet(name):      """This function Greets to      the person passed in as      parameter"""      print("Hello, " + name + ". Good morning!");    Greet("Masud");  greet("Sohel"); |

## পাইথনে function এর মধ্যে Parameter কি এবং argument কি ?

def getSum(num1, num2):

  sum = num1 + num2;

  print("Sum of the two numbers ",num1," and ",num2,"  is : ", sum);

getSum(10, 20);

লক্ষ্য করুন উপরের উদাহরণে আমরা num1 এবং num2 parameter এর জন্য দুটি আলাদা Argument যথাক্রমে 10 এবং 20 পাঠিয়েছি

## পাইথনে functions এর arguments গুলোকে কয় ভাবে receive বা গ্রহণ করা যায়?

পাইথনে দুই ভাবে argument গুলোকে receive বা গ্রহণ করা যায় :

* সরাসরি function এর ভিন্ন ভিন্ন Parameter দিয়ে ভিন্ন ভিন্ন argument গুলো রিসিভ করা যায়।
* function এর একটা নির্দিষ্ট Parameter দিয়ে সবগুলো argument গুলো রিসিভ করা যায়।

সরাসরি

1. def getSum(num1, num2):

  sum = num1 + num2;

  print("Sum of the two numbers ",num1," and ",num2,"  is : ", sum);

getSum(10, 20);

তবে আপনি চাইলে প্যারামিটার এর নাম ধরেও argument pass করতে পারেন

2.def getSum(num1, num2):

  sum = num1 + num2;

  print("Sum of the two numbers ",num1," and ",num2,"  is : ", sum);

getSum(num2=30, num1=20);

Function এর একটা নির্দিষ্ট Parameter দিয়ে সবগুলো argument গুলো রিসিভ করা যায়।

পাইথনে function এর একটা নির্দিষ্ট Parameter দিয়ে সবগুলো argument গুলো রিসিভ করতে হলে আপনাকে ফাঙ্কশনের parameter এর সামনে একটা “\*” star দিতে হবে।

def greet(\*names):

   """This function greets all

   the person in the names tuple."""

   # names is a tuple with arguments

   for name in names:

       print("Hello",name)

greet("Monir","Minhaz","Iqbal","Sharif")

## পাইথনে Variable Scope কি?

পাইথনে আপনি আপনার সমস্ত কোড জুড়ে যেকোনো জায়গায় Variable Declare বা ঘোষণা করতে পারেন। কোনো Variable যদি আপনি function এর ভিতরে ঘোষণা করেন, তাহলে ঐ Variable এর ব্যবহার Function এর মধ্যেই সীমাবদ্ধ রাখতে হবে। পাইথনে এইটাকে বলা হয় Local Scope, আবার যদি কোনো Variable ফাঙ্কশনের বাহিরে ঘোষণা করেন। তাহলে ঐ Variable সমস্ত কোড জুড়ে ব্যবহার করতে পারবেন এমনকি Funciton এর ভিতরেও ব্যবহার করতে পারবেন।

পাইথনে Variable Scope দুই ধরনের :

* local
* global

x = 10

def myFunc():

    print("Value inside function:",x)

myFunc()

তবে আপনি চাইলে variable কে ফাঙ্কশনের মধ্যে global ডিক্লেয়ার করে , ফাঙ্কশনের বাহিরে variable কে ব্যবহার করতে পারেন। চলুন উদাহরণ দিয়ে বুঝে নেয়া যাক:

def myFunc():

    global x

    x=20

    print("Value inside function:",x)

myFunc()

print("Value outside function:",x)

## local scope

def myFunc():

    x = 10

    print("Value inside function:",x)

myFunc()

print("Value outside function:",x)

## পাইথনে Anonymous এবং lambda function কি?

Anonymous শব্দের অর্থ হচ্ছে নাম বিহীন বা বেনামী, আর anonymous function মানে হচ্ছে নাম বিহীন function বা বেনামী function, সুতরাং পাইথনে যখন কোনো function declare বা ঘোষণা করা হয়, কিন্তু function টির কোনো নাম থাকেনা, পাইথনে সেই function কে anonymous function বলে। আর এই anonymous ফাঙ্কশনকেই আবার lambda function বলে।

সাধারণতঃ পাইথনে function declare করা হয় def কীওয়ার্ড ব্যবহার করে, কিন্তু পাইথনে anonymous ফাংশনগুলি lambda কীওয়ার্ড ব্যবহার করে declare করা হয়।

### পাইথন মধ্যে Lambda/anonymous ফাঙ্কশনের সিনট্যাক্স কি?

lambda arguments: expression

example:

double = lambda x: x \* 2

# Output: 10

print(double(5))

# পাইথনে print() এবং input() ফাঙ্কশনের কাজ কি?

print() ফাংশনের মাধ্যমে আউটপুট

## ইনপুট ফাংশন input()

name = input('Enter your name: ')

print("Welcome ", name)

# পাইথনে Control Structure কি?

Python Language এ Control Structure হচ্ছে প্রোগ্রাম বা Code গুলোর নিয়ন্ত্রণের কাঠামো অথবা এমন কিছু statement যা দিয়ে আপনি Program Flow Control করতে পারবেন বা Program এর পরবর্তী করণীয় কি তা নির্ধারণ করে দিতে পারবেন।

## Python Language এ Control Structure গুলো কি কি?

নিচে Control Structure গুলোর লিস্ট দেয়া হলো :

1. if
2. if…else
3. if…elif…else
4. for
5. while
6. break
7. continue
8. pass

# If the number is positive, we print an appropriate message

num = 3

if num > 0:

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

print("This is always printed.")

num = -1

if num > 0:

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

print("This is also always printed.")

if…else

age = 20

if age <=18:

    print("You're Child")

else:

    print("You're not Child")

elif Statement

#First Example

num = 0

if num > 0:

    print("Positive number")

elif num == 0:

    print("Zero")

else:

    print("Negative number")

#Second Example

num = 5

if num > 0:

    print("Positive number")

elif num == 0:

    print("Zero")

else:

    print("Negative number")

### পাইথনে loops কত প্রকার ?

Python Language এ ২ ধরণের loop আছে , সেগুলো হচ্ছে

* For
* While

## For Loop

Python Language এ for লুপ দিয়ে খুব সহজেই যেকোনো সিকোয়েন্স টাইপ অবজেক্ট যেমন list, string ইত্যাদির মধ্যে iterate করা যায়।

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

    print(letter)

একই ভাবে আমরা পাইথন range function এর সহযোগিতায় for দিয়ে 1 থেকে ১০ পর্যন্ত print করতে পারি।  
নিচের উদাহরণে লক্ষ্য করুন:

for i in range(10):

    print(i)

ঠিক একই ভাবে আমরা Python List থেকে সব value গুলো বের করে নিতে পারি

countries = ["Bangladesh", "India", "Pakistan", "Nepal"]

for country in countries:

    print(country)

### for এর সাথে else

digits = [0, 1, 5]

for i in digits:

    print(i)

else:

    print("No items left.")

## While Loop

Python Language এ while loop ও for loop এর মতো করে খুব সহজেই যেকোনো সিকোয়েন্স টাইপ অবজেক্ট যেমন list, string ইত্যাদির মধ্যে iterate করা যায়। সেক্ষেত্রে শুধু পার্থক্য হচ্ছে , while এ আগে থেকেই একটা condition set করে দেওয়া যায়। অর্থাৎ, যতক্ষণ পর্যন্ত কন্ডিশন true থাকে ততক্ষণ পর্যন্ত কোড ব্লককে ইটারেট করার জন্য পাইথনে while লুপ ব্যবহৃত হয়।

1. while লুপে প্রথমে Condition কে চেক করে। প্রোগ্রাম Loop এর Body এর মধ্যে কেবল তখনই প্রবেশ করে( যখন condition এর মান True হয়।
2. একবার ইটারেশন(iteration) এর পরে condition পুনরায় চেক হয়। condition এর মান False না হওয়া পর্যন্ত এই প্রক্রিয়া চলতেই থাকে।
3. পাইথনে while লুপের body নির্ণয়ের জন্য ইন্ডেন্টেশন ব্যবহৃত হয়।
4. Body শুরু হয় ইন্ডেন্টেশন দিয়ে এবং শেষ হয় যেখান থেকে প্রথম ইন্ডেন্টেশন শুরু হয়েছিল

str="Hello"

i=0

while i<len(str):

    print(str[i],end="\t")

    i+=1

একই ভাবে আপনি while loop দিয়ে একটি list কে iterate করতে পারবেন:

countries = ["Bangladesh", "India", "Pakistan", "Nepal"]

i=0

while i<len(countries):

    print(countries[i],end="\t")

    i+=1

while loop দিয়ে 1 থেকে 10 পর্যন্ত print:

i=1

while i<=10:

    print(i,end="\t")

    i+=1

while এর সাথে else এর ব্যবহার

এক্ষেত্রে while লুপের কন্ডিশন False হলে else অংশ সম্পাদিত হবে।

i=11

while i<=10:

    print(i,end="\t")

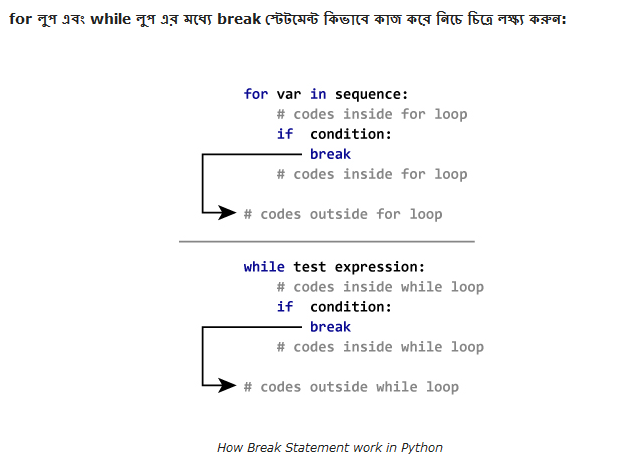
    i+=1

else:

    print("Condition Fail")

## পাইথনে break statement কি?

Python Language এ কোনো একটা নির্দিষ্ট condition true হওয়ার প্রেক্ষিতে তাৎক্ষণিক প্রোগ্রাম execution বন্ধ করা এবং program থেকে বের হওয়ার জন্য break statement ব্যবহৃত হয়।



এবার চলুন প্রথমে for লুপের মধ্যে break স্টেটমেন্টের ব্যবহার

countries = ["Bangladesh", "India", "Pakistan", "Nepal"]

for val in countries:

    if val == "Pakistan":

        print ("Pakistan not allowed in this list")

        break

    print(val)

print("The end")

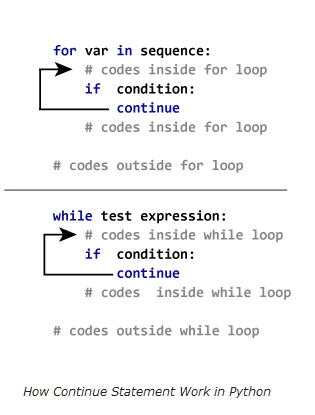
উপরের প্রোগ্রামে আমরা Country List এর প্রত্যেকটি country কে ইটারেট করতে চেয়েছিলাম। কিন্তু Break স্টেটমেন্টের ব্যবহার দেখানোর জন্য আমরা List এর মধ্যে “Pakistan” আছে কিনা চেক করেছি, “Pakistan” হলেই for লুপ থেকে বিরিত(break) নিতে চেয়েছি। তাই আমরা আমদের “Pakistan” এর পুর্ববর্তী সবগুলো country কে আউটপুট পেয়েছি। তারপরে লুপের অবসান ঘটেছে।

## Python Language এ continue statement কি?

Python Language এ কোনো একটা নির্দিষ্ট condition true হওয়ার প্রেক্ষিতে তাৎক্ষণিক প্রোগ্রাম execution skip করে পরবর্তী প্রোগ্রাম এ যাওয়ার জন্য continue statement ব্যবহৃত হয়।

### পাইথনে continue statement কিভাবে কাজ করে ?

পাইথনে continue statement সাধারণত loop এর মধ্যে একটা নির্দিষ্ট condition true হওয়ার প্রেক্ষিতে তাৎক্ষণিক প্রোগ্রাম execution skip করে পরবর্তী প্রোগ্রাম এ যাওয়ার জন্য continue statement কাজ করে। নিচের ছবিটি লক্ষ্য করুন



countries = ["Bangladesh", "India", "Pakistan", "Nepal"]

for val in countries:

    if val == "Pakistan":

        continue

    print(val)

উপরের উদাহরণে country variable এর মান যখন “Pakistan” হবে, প্রোগ্রাম তাৎক্ষণিক ভাবে তার execution skip করে পরবর্তী প্রোগ্রামে যাবে । সুতরাং এখানে ফলাফল আসবে

## পাইথনে pass স্টেটমেন্ট

Python Language এ pass একটি ফাঁকা স্টেটমেন্ট। comment এবং pass এর মধ্যে পার্থক্য হচ্ছে ইন্টারপ্রেটার কমেন্টকে সম্পূর্ণভাবে অবজ্ঞা করে কিন্তু pass কে অবজ্ঞা করে না।

countries = ["Bangladesh", "India", "Pakistan", "Nepal"] for val in countries: if val == "Pakistan": pass print("Something Wrong!") print(val)

উপরের উদাহরণে country variable এর মান যখন “Pakistan” হবে, প্রোগ্রাম তাৎক্ষণিক ভাবে null print করবে

# Python এ Operators কি?

# <http://w3programmers.com/bangla/operators/>

# পাইথনে String কি?

পাইথনে তিন ভাবে String কে রিপ্রেজেন্ট করা যায়। সে গুলো হচ্ছে Single Quotation (‘ ‘), Double Quotation (” “) এবং Three Quotation (“”” “””) এর মাধ্যমে

>>> 'This is a String with Single Quotation'

'This is a String with Single Quotation'

>>> "This is a Another String with Double Quotation"

'This is a Another String with Double Quotation'

>>> """This is Extra String with Three Quotation"""

'This is Extra String with Three Quotation'

>>>

কিছু Character কে সরাসরি একটি String এর মধ্যে ব্যবহার করা যায় না। যেমন, Double Quote (” “) দিয়ে নির্দেশ করা একটি String এর মধ্যে Double Quote (” “)থাকতে পারবেনা একইভাবে Single Quote (‘ ‘) দিয়ে নির্দেশ করা একটি String এর মধ্যে Single Quote (‘ ‘)থাকতে পারবেনা । আর এর জন্য পাইথন আপনাকে এরর প্রদর্শন করবে। আর তাই এরকম Character গুলোর সামনে একটি Backslash (\) চিহ্ন দিয়ে Escape করে দিতে হবে।

>>> 'Rizvi once said: "I\'ll be back"'

'Rizvi once said: "I\'ll be back"'

>>> "Ashiq once said: \"I'll be back\""

'Ashiq once said: "I\'ll be back"'

পাইথনে যদি একাধিক লাইন সম্বলিত কোনো String কে তিনটি করে কোটেশন (“”” “””)এর মধ্যে ডিফাইন করা হয়, তাহলে newline Character (\n) ম্যানুয়ালি লেখার দরকার পরে না

>>> """ Hello How\'re you?

Yes, I'm Fine"""

" Hello How're you?\nYes, I'm Fine"



>>> str="Hello World"

>>> print(str[0])

H

>>> print(str[0:3])

Hel

>>> print (str[0:-3])

Hello Wo

### Updating Strings (String এর একটি নির্দিষ্ট অংশ পরিবর্তন করা )

index ধরে আপনি খুব সহজে string এর বিভিন্ন অংশে পরিবর্তন আনতে পারেন, নিচের উদাহরণ লক্ষ্য করুন :

>>> str="Hello World"

>>> print ("Updated String :- ", str[:6] + 'Python')

Updated String :- Hello Python

### স্ট্রিং কনক্যাটেনেশন (Concatenation)

দুই বা ততোধিক string কে একটির সাথে আরেকটি জোড়া লাগানোর জন্য plus sign (+) ব্যবহৃত হয়, আর এটাকে বলা হয় কনক্যাটেনেশন (Concatenation).

>>> "Khalid "+"Sahin"

'Khalid Sahin'

>>> print("First Name" + ", " + "Last Name")

First Name, Last Name

>>> firstName="Masud"

>>> lastName="Alam"

>>> print(firstName + ", " + lastName)

Masud, Alam

উল্লেখ্য, মনে রাখতে হবে কোন নাম্বারের সাথে স্ট্রিং কনক্যাটেনেশন (Concatenation) করা যাবে না।

### স্ট্রিং রিপিটেশন (Repetition)

>>> print ("Hello "\*3)

Hello Hello Hello

>>> print("10 "\*3)

10 10 10

>>> print ("\*"\*10)

\*\*\*\*\*\*\*\*\*\*

### Finding Substring in a String (String এর একটি নির্দিষ্ট অংশ খোঁজা)

String এর একটি নির্দিষ্ট অংশ খোঁজার জন্য আপনি String এর in Operator ব্যবহার করতে পারেন।

>>> str="Hello World"

>>> "W" in str

True

একই ভাবে not in operator দিয়ে কোনো অংশ নাই তা নিশ্চিত করতে পারেন।

>>> str="Hello World"

>>> "Python" not in str

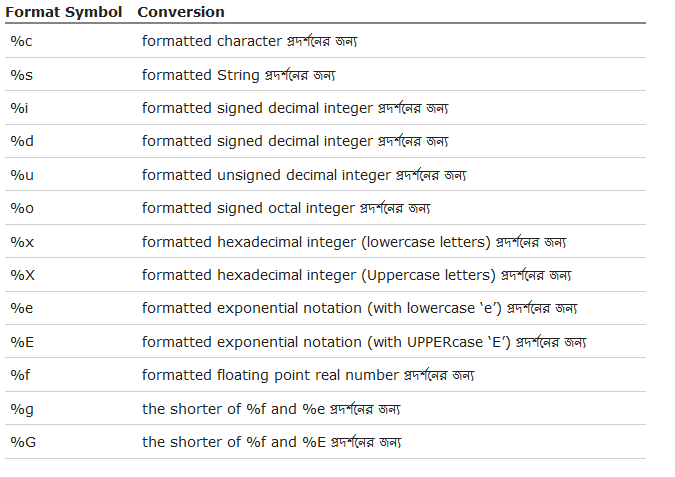
True

### String Formatting Operator

পাইথনে একটি String এর format Operator (%) টি হচ্ছে সবচেয়ে দারুন Operator , যা দিয়ে আপনি C Language এর printf function এর মতো string এর অনেক ধরণের formatting এর কাজ করতে পারেন

>>> print ("My name is %s and weight is %d kg!" % ('Masud Alam', 90))

My name is Masud Alam and weight is 90 kg!



# Python Language এ List কি?

পাইথনে List তৈরী করা খুবই সহজ। দুটো স্কয়ার ব্র্যাকেট [ ] এবং এর মধ্যে কমা দিয়ে আলাদা আলাদা Element/Item যুক্ত করে দিলেই List তৈরী হয়ে যাবে

# empty list

my\_list = []

# list of integers

my\_list = [1, 2, 3]

# list with mixed datatypes

my\_list = [1, "Hello", 3.4]

এমনকি আপনি চাইলে একটি List এর ভিতরে আরো কয়েকটি List তৈরী করে ফেলতে পারেন। যেটাকে আমরা বলতে পারি Nested List . নিচের উদাহরণ লক্ষ্য করুন :

# nested list

my\_list = ["mouse", [8, 4, 6], ['a']]

## কিভাবে List এর Element গুলোতে access করবেন?

List এর Element বা Item গুলোতে access করার জন্য আপনি index বা position ব্যবহার করতে পারেন।

my\_list = ['Hasib','Razib','Rakib','Khalid','Imran']

# Output: Hasib

print(my\_list[0])

# Output: Rakib

print(my\_list[2])

# Output: Imran

print(my\_list[4])

# Error! Only integer can be used for indexing

# my\_list[4.0]

# Nested List

n\_list = ["Happy", [2,0,1,5]]

# Nested indexing

# Output: a

print(n\_list[0][1])

# Output: 5

print(n\_list[1][3])

## Negative indexing (বিপরীত দিক থেকে indexing)

my\_list = ['Parvez','Ripon','Omran','Billal','Emran']

# Output: Emran

print(my\_list[-1])

# Output: Parvez

print(my\_list[-5])

## Python List কে Slice করা ?

my\_list = ['Parvez','Razib','Omran','Galib','Rakib','Abul','Masud','Imtiaz','Zafar Iqbal']

# elements ['Omran', 'Galib', 'Rakib']

print(my\_list[2:5])

# elements beginning to 4th

#['Parvez', 'Razib', 'Omran', 'Galib']

print(my\_list[:-5])

# elements 6th to end

#['Abul', 'Masud', 'Imtiaz', 'Zafar Iqbal']

print(my\_list[5:])

# elements beginning to end

print(my\_list[:])

## List Unpacking

List Unpacking এর মাধ্যমে একটি List (বা যেকোনো iterable) মধ্যে থাকা প্রত্যেকটি value কে আলাদা আলাদা নতুন Variable এ অ্যাসাইন করতে পারেন

>>> names=["Plabon","Shahriar","Bijon"]

>>> a,b,c=names

>>> print(a)

Plabon

>>> print(b)

Shahriar

>>> print(c)

Bijon

>>>

যদি এমন হয় যে একটি List এর মধ্যে অবস্থিত অসংখ্য ভ্যালু এর মধ্যে অল্প কিছু আলাদা আলাদা ভ্যারিয়েবলে জমা রাখতে চান, এবং বাকি গুলো একটা নির্দিষ্ট variable এ রাখতে চান। তখন নিচের মত করে যেকোনো ভ্যারিয়েবলের সামনে \* যুক্ত করে অবশিষ্ট যেকোনো সংখ্যক ভ্যালুকে এর মধ্যে জমা রাখতে পারবেন।

>>> a, b, \*c, d = [1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> print(a)

1

>>> print (b)

2

>>> print (c)

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

>>> print (d)

9

>>>

## লিস্টে পুরাতন আইটেম Update বা নতুন আইটেম add করা

# mistake values

odd = [2, 4, 6, 8]

# change the 1st item

odd[0] = 1

# Output: [1, 4, 6, 8]

print(odd)

# change 2nd to 4th items

odd[1:4] = [3, 5, 7]

# Output: [1, 3, 5, 7]

print(odd)

পাইথন লিস্টে একটি Item যোগ করার জন্য আপনি append Method ব্যবহার করতে পারেন, আবার একাধিক Item যোগ করার জন্য extend Method ব্যবহার করতে পারেন।

odd = [1, 3, 5]

odd.append(7)

# Output: [1, 3, 5, 7]

print(odd)

odd.extend([9, 11, 13])

# Output: [1, 3, 5, 7, 9, 11, 13]

print(odd)

এছাড়াও আপনি দুইটি list কে combine করার জন্য plus (+) operator ব্যবহার করতে পারেন, যেটাকে আবার concatenation ও বলা হয়ে থাকে।

odd = [1, 3, 5]

# Output: [1, 3, 5, 9, 7, 5]

print(odd + [9, 7, 5])

একইভাবে লিস্টের মধ্যে Item গুলোকে repeat করার জন্য Multiplication (\*) operator ব্যবহার করতে পারেন। #Output: ["re", "re", "re"]

print(["re"] \* 3)

পাইথনে লিস্টে আপনি যদি একটি নির্দিষ্ট location এ কিছু add করতে চান , সেক্ষেত্রে আপনাকে insert Method ব্যবহার করতে হবে, লক্ষ্য করুন odd = [1, 9]

odd.insert(1,3)

# Output: [1, 3, 9]

print(odd)

odd[2:2] = [5, 7]

# Output: [1, 3, 5, 7, 9]

print(odd)

## List থেকে কোনো কিছু remove বা Delete করা

my\_list = ['p','r','o','b','l','e','m']

# delete one item

del my\_list[2]

# Output: ['p', 'r', 'b', 'l', 'e', 'm']

print(my\_list)

# delete multiple items

del my\_list[1:5]

# Output: ['p', 'm']

print(my\_list)

# delete entire list

del my\_list

# Error: List not defined

print(my\_list)

পাইথনে লিস্টের Item ধরে remove করতে চাইলে আপনাকে remove method ব্যবহার করতে হবে, একইভাবে Item এর key ধরে remove করতে চাইলে আপনাকে pop Method ব্যবহার করতে হবে। আর লিস্টকে খালি করে দেওয়ার জন্য আপনি clear Method ব্যবহার করতে পারেন। নিচের উদাহরণ দেখুন :

my\_list = ['p','r','o','b','l','e','m']

my\_list.remove('p')

# Output: ['r', 'o', 'b', 'l', 'e', 'm']

print(my\_list)

# Output: 'o'

print(my\_list.pop(1))

# Output: ['r', 'b', 'l', 'e', 'm']

print(my\_list)

# Output: 'm'

print(my\_list.pop())

# Output: ['r', 'b', 'l', 'e']

print(my\_list)

my\_list.clear()

# Output: []

print(my\_list)

এছাড়াও একটি empty List assign করার মাধমেও আপনি লিস্টের Item রিমুভ করতে পারেন।

>>> my\_list = ['p','r','o','b','l','e','m']

>>> my\_list[2:3] = []

>>> my\_list

['p', 'r', 'b', 'l', 'e', 'm']

>>> my\_list[2:5] = []

>>> my\_list

['p', 'r', 'm']

লিস্টের কোনো Item এর index বের করার জন্য আপনি index Method ব্যবহার করতে পারেন। অবশ্য index Method আপনাকে লিস্টের First Matched Item টি প্রদর্শিত করবে। একইভাবে লিস্টের Item সংখ্যা count করার জন্য count Method ব্যবহার করতে পারেন। লিস্টকে Ascending আকারে প্রদর্শন করার জন্য sort Method এবং Reverse অর্ডারে দেখানোর জন্য reverse Method ব্যবহার করতে পারেন। আর লিস্টকে copy করার জন্য copy Method ব্যবহার করতে পারেন

my\_list = [3, 8, 1, 6, 0, 8, 4]

# Output: 1

print(my\_list.index(8))

# Output: 2

print(my\_list.count(8))

my\_list.sort()

# Output: [0, 1, 3, 4, 6, 8, 8]

print(my\_list)

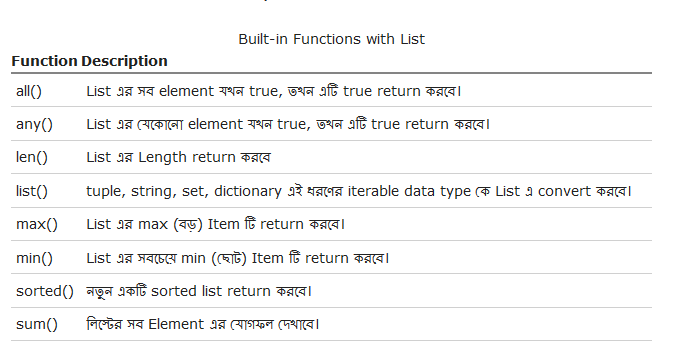
my\_list.reverse()

# Output: [8, 8, 6, 4, 3, 1, 0]

print(my\_list)

my\_list.copy()

## List এর কিছু Built-in Function List



>>> my\_list=[2,1,3,6,5,8,7,9,10,21]

>>> len(my\_list)

10

>>> max(my\_list)

21

>>> min(my\_list)

1

>>> sum(my\_list)

72

>>> sorted(my\_list)

[1, 2, 3, 5, 6, 7, 8, 9, 10, 21]

>>> any(my\_list)

True

>>> all(my\_list)

True

>>>

## List এ for loop এর ব্যবহার :

for fruit in ['apple','banana','mango']:

    print("I like",fruit)

## List এ কোনো Item আছে কিনা তা যাচাই করা :

my\_list = ['p','r','o','b','l','e','m']

# Output: True

print('p' in my\_list)

# Output: False

print('a' in my\_list)

# Output: True

print('c' not in my\_list)

List এর প্রত্যেকটি element এর সাথে power যোগ করে নতুন List তৈরির আরো একটি উদাহরণ : pow2 = [2 \*\* x for x in range(10)]

# Output: [1, 2, 4, 8, 16, 32, 64, 128, 256, 512]

print(pow2)

## Python List Comprehension

ধরুন আমরা “human” word টির letter গুলোকে আলাদা করে একটি List এর Item হিসেবে রাখবো। তাহলে কি করতে হবে? সহজ উত্তর for loop ব্যবহার করলেই হয়, তাহলে চলুন for loop দিয়ে কাজটি করা যাক :

h\_letters = []

for letter in 'human':

    h\_letters.append(letter)

print(h\_letters)

['h','u','m','a','n']

আর পাইথনে List Comprehension ব্যবহার করে , আপনি এই কাজটি আরো সহজেই করতে পারতেন। আর মূলতঃ পাইথনে List Comprehension হচ্ছে existing List এর উপর ভিত্তি করে আরো সহজে List তৈরী করার উপায়। চলুন একটি উদাহরণ দিয়ে ব্যাপারটা বুঝা যাক :

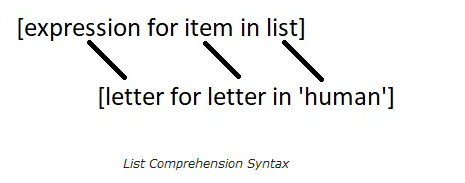
h\_letters = [ letter for letter in 'human' ]

print( h\_letters)

['h', 'u', 'm', 'a', 'n']

## List Comprehension এর Syntax

[expression for item in list]



## List Comprehension এর মধ্যে if Condition add করা

number\_list = [ x for x in range(20) if x % 2 == 0]

print(number\_list)

## List Comprehension এর মধ্যে Nested if Condition add করা

num\_list = [y for y in range(100) if y % 2 == 0 if y % 5 == 0]

print(num\_list)

## List Comprehension এর মধ্যে if else Condition add করা

obj = ["Even" if i%2==0 else "Odd" for i in range(10)]

print(obj)

# Python Language এ Tuple কি?

Python Language এ লিস্টের মতই আরেকটি ডাটা স্ট্রাকচার হচ্ছে Tuple. পার্থক্যটি হচ্ছে আপনি চাইলে List এর যেকোনো value পরিবর্তন করতে পারবেন কিন্তু tuple এর value পরিবর্তন করতে পারবেন না। অর্থাৎ , Tuple কে লিস্টের মত করে ইচ্ছামত পরিবর্তন, পরিবর্ধন করতে পারবেন না। Tuple এ প্রত্যেকটি Element/Item এর জন্য একটি number assign করা হয় যাকে index বা position বলা যায়। Tuple অনেকটা C, C++, Java এবং Python Tuple এর মতো। Tuple এ প্রথম index zero (0), তারপর 1 এবং এরপর ক্রমিক আকারে বাড়তে থাকে।আবার, লিস্ট যেমন তৈরি করতে হয় দুটো [] ব্র্যাকেট দিয়ে কিন্তু টাপল তৈরি করতে হয় () দিয়ে (যদিও ব্র্যাকেট ছাড়াও শুধু কমা চিহ্ন দিয়ে ভ্যালু গুলোকে আলাদা করেও টাপল তৈরি করা যায়)। মূলত , Data কে সুরক্ষিত রাখতেই টাপল(Tuple) ব্যবহৃত হয় এবং এটি Dynamically পরিবর্তন না হওয়ায় List এর তুলনায় দ্রুত কাজ করে।

## পাইথনে কিভাবে Tuple তৈরী করবেন ?

পাইথনে Tuple তৈরী করা খুবই সহজ। দুটো প্যারেন্থেসিস ( ) এবং এর মধ্যে কমা দিয়ে আলাদা আলাদা Element/Item যুক্ত করে দিলেই (যদিও ব্র্যাকেট ছাড়াও শুধু কমা চিহ্ন দিয়ে ভ্যালু গুলোকে আলাদা করেও টাপল তৈরি করা যায়) Tuple তৈরী হয়ে যাবে। চলুন একসাথে কয়েকটি Tuple তৈরী করে ফেলি :

# empty tuple

# Output: ()

my\_tuple = ()

print(my\_tuple)

# tuple having integers

# Output: (1, 2, 3)

my\_tuple = (1, 2, 3)

print(my\_tuple)

# tuple with mixed datatypes

# Output: (1, "Hello", 3.4)

my\_tuple = (1, "Hello", 3.4)

print(my\_tuple)

# nested tuple

# Output: ("mouse", [8, 4, 6], (1, 2, 3))

my\_tuple = ("mouse", [8, 4, 6], (1, 2, 3))

print(my\_tuple)

# tuple can be created without parentheses

# also called tuple packing

# Output: 3, 4.6, "dog"

my\_tuple = 3, 4.6, "dog"

print(my\_tuple)

# tuple unpacking is also possible

# Output:

# 3

# 4.6

# dog

a, b, c = my\_tuple

print(a)

print(b)

print(c)

## কিভাবে Tuple এর Element গুলোতে access করবেন?

Tuple এর Element বা Item গুলোতে access করার জন্য আপনি List এর মতোই index বা position ব্যবহার করতে পারেন।

my\_tuple = ('Parvez','Erfan','Ramjan','Maksud','Imran','Taher')

# Output: 'Parvez'

print(my\_tuple[0])

# Output: 'Taher'

print(my\_tuple[5])

#Nested Tuple

n\_tuple = ("mouse", [8, 4, 6], (1, 2, 3))

# nested index

# Output: 's'

print(n\_tuple[0][3])

# nested index

# Output: 4

print(n\_tuple[1][1])

## negative indexing (বিপরীত দিক থেকে indexing)

my\_tuple = ('Pervez','Emran','Rimon','Momtaz','Irfan','Taher')

# Output: 'Taher'

print(my\_tuple[-1])

# Output: 'Parvez'

print(my\_tuple[-6])

## Python Tuple কে Slice করা ?

my\_tuple = ('Parvez','Rakib','Omran','Galib','Riad','Anwar','Mahbub','Imran','Zaber')

# elements 2nd to 4th

# Output: ('Rakib', 'Omran', 'Galib')

print(my\_tuple[1:4])

# elements beginning to 2nd

# Output: ('Parvez', 'Rakib')

print(my\_tuple[:-7])

# elements 8th to end

# Output: ('Imran', 'Zaber')

print(my\_tuple[7:])

# elements beginning to end

# Output: ('Parvez','Rakib','Omran','Galib','Riad','Anwar','Mahbub','Imran','Zaber')

print(my\_tuple[:])

## Tuple Unpacking

Tuple Unpacking এর মাধ্যমে একটি টাপলের (বা যেকোনো iterable) মধ্যে থাকা প্রত্যেকটি value কে আলাদা আলাদা নতুন Variable এ অ্যাসাইন করতে পারেন এক লাইন কোড লিখেই।

>>> names=("Plabon","Shahriar","Bijon")

>>> a,b,c=names

>>> print (a)

Plabon

>>> print (b)

Shahriar

>>> print (c)

Bijon

>>>

যদি এমন হয় যে একটি Tuple এর মধ্যে অবস্থিত অসংখ্য ভ্যালু এর মধ্যে অল্প কিছু আলাদা আলাদা ভ্যারিয়েবলে জমা রাখতে চান, এবং বাকি গুলো একটা নির্দিষ্ট variable এ রাখতে চান। তখন নিচের মত করে যেকোনো ভ্যারিয়েবলের সামনে \* যুক্ত করে অবশিষ্ট যেকোনো সংখ্যক ভ্যালুকে এর মধ্যে জমা রাখতে পারবেন।

>>> a, b, \*c, d = (1, 2, 3, 4, 5, 6, 7, 8, 9)

>>> print(a)

1

>>> print (b)

2

>>> print (c)

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

>>> print (d)

9

>>>

## Tuple এ পুরাতন আইটেম Update বা নতুন আইটেম add করা :

my\_tuple = (4, 2, 3, [6, 5])

# we cannot change an element

# If you uncomment line 8

# you will get an error:

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

#my\_tuple[1] = 9

# but item of mutable element can be changed

# Output: (4, 2, 3, [9, 5])

my\_tuple[3][0] = 9

print(my\_tuple)

# tuples can be reassigned

# Output: ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')

my\_tuple = ('p','r','o','g','r','a','m','i','z')

print(my\_tuple)

## দুইটি tuple কে combine করা

আপনি দুইটি tuple কে combine করার জন্য plus (+) operator ব্যবহার করতে পারেন, যেটাকে আবার concatenation ও বলা হয়ে থাকে।# Concatenation

# Output: (1, 2, 3, 4, 5, 6)

print((1, 2, 3) + (4, 5, 6))

## tuple Item কে repeat করা

একইভাবে Tuple এর মধ্যে Item গুলোকে repeat করার জন্য Multiplication (\*) operator ব্যবহার করতে পারেন।# Repeat

# Output: ('Repeat', 'Repeat', 'Repeat')

print(("Repeat",) \* 3)

## Tuple থেকে কোনো কিছু remove বা Delete করা

পাইথনে Tuple এর মধ্যে কোনো নির্দিষ্ট Item কে রিমুভ করতে না পারলেও আপনি পুরো tuple কেই ডিলিট বা রিমোভ করার জন্য del কীওয়ার্ডটি ব্যবহার করতে পারেন। নিচের উদাহরণ দেখুন :

my\_tuple = ('Parvez','Robin','Omran','Galib','Raihan','Amanullah','Mahfuz','Imran','Zafreen')

# can't delete items

# if you uncomment line 8,

# you will get an error:

# TypeError: 'tuple' object doesn't support item deletion

#del my\_tuple[3]

# can delete entire tuple

# NameError: name 'my\_tuple' is not defined

del my\_tuple

my\_tuple

## পাইথনে Tuple এর Item সংখ্যা Count

পাইথনে Tuple এর Item গুলোর মধ্যে কোনো নির্দিষ্ট Item সংখ্যা কত, তা গণনা করার জন্য আপনি count Method ব্যবহার করতে পারেন। নিচের উদাহরণ দেখুন :

my\_tuple = ('a','p','p','l','e',)

# Count

# Output: 2

print(my\_tuple.count('p'))

পাইথনে Tuple এর Item গুলোর মধ্যে কোনো নির্দিষ্ট Item এর index কত, তা জানার জন্য আপনি index Method ব্যবহার করতে পারেন। নিচের উদাহরণ দেখুন : my\_tuple = ('a','p','p','l','e',)

# Index

# Output: 3

print(my\_tuple.index('l'))

## Tuple এ কোনো Item আছে কিনা তা যাচাই করা :

my\_tuple = ('a','p','p','l','e',)

# In operation

# Output: True

print('a' in my\_tuple)

# Output: False

print('b' in my\_tuple)

# Not in operation

# Output: True

print('g' not in my\_tuple)

## Iterate to Tuple’s Items

পাইথনের for loop দিয়ে আপনি item গুলোকে iterate করতে পারি।

# Output:

# Hello Jahir

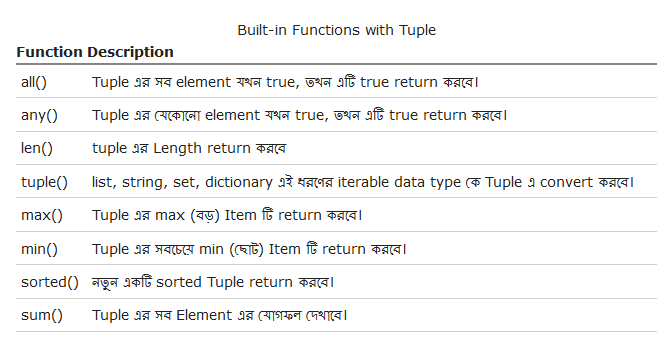
# Hello Karim

for name in ('Jahir','Karim'):

     print("Hello",name)

## Tuple এর কিছু Built-in Function List

Tuple নিয়ে বিভিন্ন কাজ করার জন্য সাধারণ কিছু function নিচে সেগুলোর বিস্তারিত দেওয়া হলো।

>>> my\_tuple=[2,1,3,6,5,8,7,9,10,21]

>>> len(my\_tuple)

10

>>> max(my\_tuple)

21

>>> min(my\_tuple)

1

>>> sum(my\_tuple)

72

>>> sorted(my\_tuple)

[1, 2, 3, 5, 6, 7, 8, 9, 10, 21]

>>> any(my\_tuple)

True

>>> all(my\_tuple)

True

>>>

# Python Language এ Dictionary কি?

Python Language এ List এর মতই আরেকটি ডাটা স্ট্রাকচার হচ্ছে Dictionary., পার্থক্যটা হচ্ছে List এ থাকে বাই ডিফল্ট number index আর Dictionary তে থাকে Key. List এর index গুলো নিজে নিজেই তৈরী হয়, আর Dictionary এর key গুলোকে Manually বসিয়ে দিতে হয়। আরো সহজভাবে বলা যায় , Dictionary তে Data গুলো থাকে জোড়ায় জোড়ায়। আর জোড়ার মধ্যে প্রথম জিনিসটিকে বলে কি (key) আর দ্বিতীয় জিনিসটিকে বলে ভ্যালু (value)। Dictionary তে key হিসেবে string, integer এমনকি tuple ও ব্যবহার করতে পারবেন। List এ যেমন index নম্বর একটা value কে নির্দেশ করে ঠিক তেমনি করে Dictionary তে একটা কী (Key) একটা value কে নির্দেশ করে। এই value টা String, Integer, Float, List, Tuple ইত্যাদি যেকোন টাইপেরই হতে পারে। একটা Dictionary এর প্রতিটা key ইউনিক হতে হয়, অর্থাৎ , একটার সাথে অপরটা মিলতে পারবে না। তবে একই value কয়েকবার থাকতে পারে

## Dictionary কিভাবে তৈরী করা হয়?

(curly braces) { } ব্র্যাকেট এর মধ্যে ( : ) কোলন চিহ্ন দিয়ে key-value জোড় তৈরি করে এবং প্রত্যেক জোড় কে কমা ( , ) দিয়ে আলাদা করে একটি Dictionary তৈরি করা যায় । এ ছাড়াও আপনি dict() function দিয়েও Dictionary তৈরী করতে পারেন।

# empty dictionary

my\_dict = {}

# dictionary with integer keys

my\_dict = {1: 'apple', 2: 'ball'}

# dictionary with mixed keys

my\_dict = {'name': 'Maruf', 1: [2, 4, 3]}

# using dict()

my\_dict = dict({1:'apple', 2:'ball'})

# from sequence having each item as a pair

my\_dict = dict([(1,'apple'), (2,'ball')])

## Dictionary এর Item গুলোতে কিভাবে access করবেন?

Dictionary এর প্রত্যেকটি Element কে access করার নিয়ম List এর মতই। List এ যেমন [ ] থার্ড ব্র্যাকেট এর মধ্যে index দিয়ে উক্ত ইনডেক্সের value print করতে হয়, তেমনি Dictionary এর ক্ষেত্রেও থার্ড ব্র্যাকেট এর মধ্যে ইনডেক্সের যায়গায় key ব্যবহার করে উক্ত key এর value তে access করতে হয়। এ ছাড়া আপনি get() Method দিয়েও Dictionary Item গুলোতে access করতে পারেন।

my\_dict = {'name':'Jaber', 'age': 26}

# Output: Jaber

print(my\_dict['name'])

# Output: 26

print(my\_dict.get('age'))

# Trying to access keys which doesn't exist throws error

# my\_dict.get('address')

# my\_dict['address']

## কিভাবে Dictionary তে পুরাতন Item Update বা নতুন Item add করবেন?

Dictionary তে কোন Item কে Insert বা Update করা খুবই সহজ কাজ। আর এর জন্য শুধু key দিয়ে তাতে নতুন ডাটা অ্যাসাইন করলেই হবে। যদি ঐ key ইতিমধ্যে থেকে থাকে তাহলে Dictionary Update হয়ে যাবে। আর যদি key টি পূর্ব থেকে না থেকে থাকে , তাহলে এটি নতুন Item হিসেবে Dictionary তে add হয়ে যাবে।

my\_dict = {'name':'Jahir', 'age': 26}

# update value

my\_dict['age'] = 27

#Output: {'age': 27, 'name': 'Jahir'}

print(my\_dict)

# add item

my\_dict['address'] = 'Dhaka,Bangladesh'

# Output: {'address': 'Dhaka,Bangladesh', 'age': 27, 'name': 'Jahir'}

print(my\_dict)

## Dictionary থেকে কোনো কিছু remove বা Delete করা

পাইথনে Dictionary থেকে এক বা একাধিক এমনকি পুরো Dictionary ডিলিট বা রিমোভ করে দিতে পারবেন। আর জন্য আছে pop(), popitem(), clear() মেথড । pop() মেথড মূলত একটি নির্দিষ্ট key ধরে যেকোনো আইটেম কে রিমুভ বা রিটার্ন করার জন্য ব্যবহার হয়। আর popitem() দিয়ে random বা arbitrary আইটেম রিমুভ অথবা রিটার্ন করতে পারবেন। আর সব আইটেম একসাথে রিমুভ করার জন্য আপনি clear() Method ব্যবহার করতে পারবেন। এছাড়াও আপনি del কীওয়ার্ড দিয়ে একটি নির্দিষ্ট dictionary item অথবা পুরো dictionary কে Delete করে দিতে পারেন।

# create a dictionary

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

# remove a particular item

# Output: 16

print(squares.pop(4))

# Output: {1: 1, 2: 4, 3: 9, 5: 25}

print(squares)

# remove an arbitrary item

# Output: (1, 1)

print(squares.popitem())

# Output: {2: 4, 3: 9, 5: 25}

print(squares)

# delete a particular item

del squares[5]

# Output: {2: 4, 3: 9}

print(squares)

# remove all items

squares.clear()

# Output: {}

print(squares)

# delete the dictionary itself

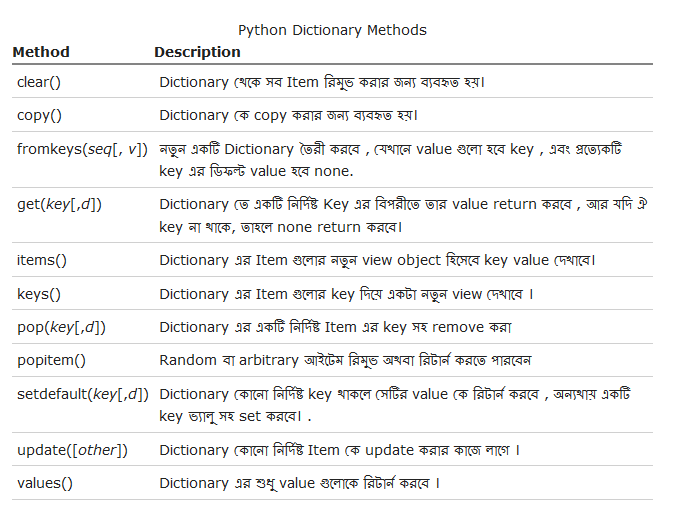
del squares

# Throws Error

# print(squares)

## Python Dictionary Methods

Dictionary নিয়ে বিভিন্ন কাজ করার জন্য Method গুলোর List নিচে বিস্তারিত দেওয়া হলো। যার অনেকগুলোরই বিস্তারিত উপরে আলোচনা হয়েছে।



# random sales dictionary

sales = { 'apple': 2, 'orange': 3, 'grapes': 4 }

#dict\_values([2, 3, 4])

print(sales.values())

person = {'name': 'Masud Alam', 'age': 22}

age = person.setdefault('age')

#person =  {'age': 22, 'name': 'Masud Alam'}

print('person = ',person)

#Age =  22

print('Age = ',age)

person = {'name': 'Masud Alam', 'age': 22, 'salary': 3500.0}

#dict\_keys(['age', 'name', 'salary'])

print(person.keys())

#dict\_keys([])

empty\_dict = {}

print(empty\_dict.keys())

# random sales dictionary

sales = { 'apple': 2, 'orange': 3, 'grapes': 4 }

#dict\_items([('apple', 2), ('orange', 3), ('grapes', 4)])

print(sales.items())

# vowels keys

keys = {'a', 'e', 'i', 'o', 'u' }

vowels = dict.fromkeys(keys)

#{'i': None, 'e': None, 'u': None, 'o': None, 'a': None}

print(vowels)

original = {1:'one', 2:'two'}

new = original.copy()

#Orignal:  {1: 'one', 2: 'two'}

print('Orignal: ', original)

#New:  {1: 'one', 2: 'two'}

print('New: ', new)

## Python Dictionary Comprehension

একটা iterable List থেকে নতুন একটি Dictionary তে রূপান্তর করার পদ্ধতিকে Dictionary Comprehension বলা হয়।

squares = {}

for x in range(6):

   squares[x] = x\*x

#{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

 print(squares)

আর এটাকে আমরা Dictionary তে রূপান্তর করার জন্য নিচের মতো code করবো।

squares = {x: x\*x for x in range(6)}

# Output: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

print(squares)

এমনকি আমরা dictionary comprehesion এর ভিতরে if ও ব্যবহার করতে পারি।

odd\_squares = {x: x\*x for x in range(11) if x%2 == 1}

# Output: {1: 1, 3: 9, 5: 25, 7: 49, 9: 81}

print(odd\_squares)

## অন্যান্য dictionary Operations

Dictionary এর ভিতরে কোনো key আছে কিনা তা চেক করার জন্য আমরা in Operator ব্যবহার করতে পারি।

squares = {1: 1, 3: 9, 5: 25, 7: 49, 9: 81}

# Output: True

print(1 in squares)

# Output: True

print(2 not in squares)

# membership tests for key only not value

# Output: False

print(49 in squares)

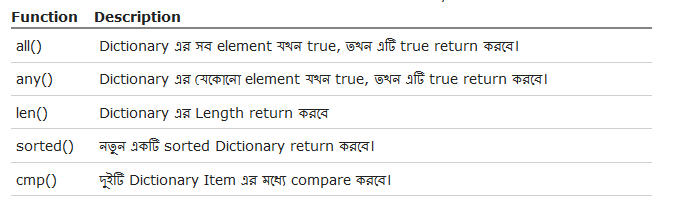
Dictionary এর ভিতরে প্রত্যেকটি key কে for loop দিয়ে iterate করতে পারি।

squares = {1: 1, 3: 9, 5: 25, 7: 49, 9: 81}

for i in squares:

    print(squares[i])

## Dictionary এর কিছু Built-in Function List



squares = {1: 1, 3: 9, 5: 25, 7: 49, 9: 81}

# Output: 5

print(len(squares))

# Output: [1, 3, 5, 7, 9]

print(sorted(squares))

## পাইথনে Nested Dictionary কি?

পাইথনে যখন একটি Dictionary এর মধ্যে একাধিক Dictionary থাকে তখন এটাকে Nested Dictionary বলে

people = {1: {'name': 'Borhan', 'age': '27', 'sex': 'Male'},

          2: {'name': 'Nahida', 'age': '22', 'sex': 'Female'}}

print(people)

উপরের কোডটি run করলে নিচের মতো ফলাফল দেখাবে : {

1: {'age': '27', 'sex': 'Male', 'name': 'Borhan'},

2: {'age': '22', 'sex': 'Female', 'name': 'Nahida'}

}

ব্যাখ্যা: উপরের প্রোগ্রামে $people হচ্ছে একটা Nested Dictionary, যার মধ্যে আবার index 1 এবং ২ তে দুইটি dictionary add করা হয়েছে।

## Nested Dictionary এর Element গুলোতে Access

Nested Dictionary এর Element গুলোতে Access করতে হলে আপনাকে [ ] ব্যবহার করতে হবে, চলুন নিচে উদাহরণ দিয়ে বুঝে নেয়া যাক:

people = {1: {'name': 'Borhan', 'age': '27', 'sex': 'Male'},

          2: {'name': 'Nahida', 'age': '22', 'sex': 'Female'}}

print(people[1]['name'])

print(people[1]['age'])

print(people[1]['sex'])

## Nested Dictionary তে Element Add অথবা update করা :

Nested Dictionary তে Element Add অথবা update করতে হলে আপনাকে প্রথমে একটা নতুন Dictionary তৈরী করে নিতে হবে, তারপর সেখানে নতুন Element Add করতে পারবেন।

people = {1: {'name': 'John', 'age': '27', 'sex': 'Male'},

          2: {'name': 'Marie', 'age': '22', 'sex': 'Female'}}

people[3] = {}

people[3]['name'] = 'Farhana'

people[3]['age'] = '24'

people[3]['sex'] = 'Female'

people[3]['married'] = 'No'

print(people[3])

উপরের কোডটি run করলে নিচের মতো ফলাফল দেখাবে :

{'name': 'Farhana', 'age': '24', 'sex': 'Female', 'married': 'No'}

তবে নিচের উদাহরণের মতো করেও Nested Dictionary তে Element Add অথবা update করতে পারবেন :

people = {1: {'name': 'Borhan', 'age': '27', 'sex': 'Male'},

          2: {'name': 'Moumita', 'age': '22', 'sex': 'Female'},

          3: {'name': 'Farhana', 'age': '24', 'sex': 'Female', 'married': 'No'}}

people[4] = {'name': 'Polash', 'age': '29', 'sex': 'Male', 'married': 'Yes'}

print(people[4])

উপরের কোডটি run করলে নিচের মতো ফলাফল দেখাবে :

{'name': 'Peter', 'age': '29', 'sex': 'Male', 'married': 'Yes'}

## Nested Dictionary থেকে Element Delete করা

Nested Dictionary থেকে Element Delete করতে হলে আপনাকে del keyword টি ব্যবহার করতে হবে। চলুন উদাহরণ দিয়ে বুঝে নেয়া যাক :

people = {1: {'name': 'Borhan', 'age': '27', 'sex': 'Male'},

          2: {'name': 'Farhana', 'age': '22', 'sex': 'Female'},

          3: {'name': 'Moumita', 'age': '24', 'sex': 'Female', 'married': 'No'},

          4: {'name': 'Polash', 'age': '29', 'sex': 'Male', 'married': 'Yes'}}

del people[3]['married']

del people[4]['married']

print(people[3])

print(people[4])

{'name': 'Moumita', 'age': '24', 'sex': 'Female'}

{'name': 'Polash', 'age': '29', 'sex': 'Male'}

আবার আপনি চাইলে পুরো Dictionary ই Delete করে দিতে পারেন

people = {1: {'name': 'Borhan', 'age': '27', 'sex': 'Male'},

          2: {'name': 'Farhana', 'age': '22', 'sex': 'Female'},

          3: {'name': 'Moumita', 'age': '24', 'sex': 'Female'},

          4: {'name': 'Peter', 'age': '29', 'sex': 'Male'}}

del people[3], people[4]

print(people)

উপরের কোডটি run করলে নিচের মতো ফলাফল দেখাবে :

{1: {'name': 'Borhan', 'age': '27', 'sex': 'Male'}, 2: {'name': 'Farhana', 'age': '22', 'sex': 'Female'}}

## Nested Dictionary এর মধ্যে Iterate

for loop ব্যবহার করে আপনি Nested Dictionary এর প্রত্যেকটি Element এ iterate করতে পারেন। নিচের উদাহরণ দেখুন :

people = {1: {'Name': 'Jahir', 'Age': '27', 'Sex': 'Male'},

          2: {'Name': 'Moumita', 'Age': '22', 'Sex': 'Female'}}

for p\_id, p\_info in people.items():

    print("\nPerson ID:", p\_id)

    for key in p\_info:

        print(key + ':', p\_info[key])

# Python Language এ Set কি?

Python Language এ List এবং Tuple এর মতই আরেকটি ডাটা স্ট্রাকচার হচ্ছে Set. পার্থক্যটি হচ্ছে আপনি চাইলে List এবং Tuple এর মধ্যে duplicate value রাখতে পারবেন কিন্তু Set এ কোনো duplicate Value রাখতে পারবেন না। আর List এবং Tuple এর value গুলো index আকারে সাজানো থাকে কিন্তু Set এর Value গুলো indexing আকারে থাকেনা বা indexing করা যায় না। সেটা অবশ্য লাগেওনা, কারণ Set এর প্রত্যেকটি value ই ইউনিক। মূলতঃ গাণিতিক কাজ যেমন union, intersection এবং symmetric difference ইত্যাদি বের করার জন্য Set ব্যবহৃত হয়।

## Set কিভাবে তৈরী করা হয়?

set তৈরী করতে হলে আপনাকে (curly braces) { } ব্র্যাকেট অথবা set ফাংশন ব্যবহার করতে হবে , তবে ফাকা set তৈরি করার সময় { } ব্যবহার করা যাবে না কারণ এটা ফাকা ডিকশনারি তৈরি করার সাথে কনফ্লিক্ট করবে। বরং set() function ব্যবহার করে ফাকা সেট তৈরি করতে হয়

# set of integers

my\_set = {1, 2, 3}

print(my\_set)

# set of mixed datatypes

my\_set = {1.0, "Hello", (1, 2, 3)}

print(my\_set)

set এর মধ্যে পরিবর্তন করা যাবে (mutable) এইরকম Data Type যেমন List এবং Dictionary রাখা যাবেনা।

# set do not have duplicates

# Output: {1, 2, 3, 4}

my\_set = {1,2,3,4,3,2}

print(my\_set)

# set cannot have mutable items

# here [3, 4] is a mutable list

# If you uncomment line #12,

# this will cause an error.

# TypeError: unhashable type: 'list'

#my\_set = {1, 2, [3, 4]}

# we can make set from a list

# Output: {1, 2, 3}

my\_set = set([1,2,3,2])

print(my\_set)

## কিভাবে পাইথন Set এ চেঞ্জ আনবেন?

আমরা আগেই জেনেছি , set এ কোনো indexing থাকেনা এবং এর value গুলো unordered অর্থাৎ যেইভাবে ইচ্ছা সেইভাবে রাখা যায় , শুধু duplicate না হলেই হলো। আর যেহেতু set এ কোনো index থাকেনা তাই একে আপনি index ধরে কোনো রকম access অথবা slicing করতে পারবেন না। তবে আপনি চাইলে add() function দিয়ে set এর মধ্যে নতুন value add করতে পারবেন এবং update() function দিয়ে আগের value কে update করতে পারবেন। তাছাড়া update Function দিয়ে আপনি অন্যান্য data type যেমন list, tuple কে set এ পরিবর্তন করতে পারবেন। চলুন কয়েকটি উদাহরণ দিয়ে বুঝা যাক:

# initialize my\_set

my\_set = {1,3}

print(my\_set)

# if you uncomment line 9,

# you will get an error

# TypeError: 'set' object does not support indexing

#my\_set[0]

# add an element

# Output: {1, 2, 3}

my\_set.add(2)

print(my\_set)

# add multiple elements

# Output: {1, 2, 3, 4}

my\_set.update([2,3,4])

print(my\_set)

# add list and set

# Output: {1, 2, 3, 4, 5, 6, 8}

my\_set.update([4,5], {1,6,8})

print(my\_set)

এখানে আপনি যদি লাইন ৯ কে uncomment করেন , তাহলে পাইথন আপনাকে error প্রদর্শন করবে। কারণ set এ indexing support করেনা।

## Set থেকে কোনো element কিভাবে remove করবেন?

Set থেকে কোনো element কে remove করার জন্য discard() function এবং remove() function ব্যবহৃত হয়। পার্থক্য হচ্ছে discard() function দিয়ে remove করা element যদি Set এ না থাকে পাইথন আপনাকে কোনো error প্রদর্শন করবেনা, এবং set অপরিবর্তনীয় থাকবে, অন্যদিকে remove() function দিয়ে কোনো কিছু remove করা কালীন সেই element যদি set এ না থাকে, তাহলে Python আপনাকে error প্রদর্শন করবে। নিচের উদাহরণ গুলো লক্ষ্য করুন :

# initialize my\_set

my\_set = {1, 3, 4, 5, 6}

print(my\_set)

# discard an element

# Output: {1, 3, 5, 6}

my\_set.discard(4)

print(my\_set)

# remove an element

# Output: {1, 3, 5}

my\_set.remove(6)

print(my\_set)

# discard an element

# not present in my\_set

# Output: {1, 3, 5}

my\_set.discard(2)

print(my\_set)

# remove an element

# not present in my\_set

# If you uncomment line 27,

# you will get an error.

# Output: KeyError: 2

#my\_set.remove(2)

উল্লেখ্য , এখানে আপনি যদি লাইন ২৭ কে uncomment করেন , তাহলে পাইথন আপনাকে error প্রদর্শন করবে। কারণ remove() function দিয়ে যেটি রিমুভ করার চেষ্টা করতেছেন সেটি Set এর মধ্যে নেই।

একইভাবে আপনি চাইলে Set থেকে কোনো element কে pop() function দিয়ে remove অথবা return করতে পারবেন। আর যেহেতু Python Set এ কোনো index থাকেনা , তাই আপনি বুঝতে পারবেন না , কোন আইটেম টি remove হয়েছে। অর্থাৎ randomly Item remove হবে।

# initialize my\_set

# Output: set of unique elements

my\_set = set("HelloWorld")

print(my\_set)

# pop an element

# Output: random element

print(my\_set.pop())

# pop another element

# Output: random element

my\_set.pop()

print(my\_set)

পাইথনের clear() function দিয়ে আপনি পুরো set কেই remove করে দিতে পারেন। নিচের উদাহরণ দেখুন :

# initialize my\_set

# Output: set of unique elements

my\_set = set("HelloWorld")

# clear my\_set

#Output: set()

my\_set.clear()

print(my\_set)

## SET দিয়ে আরো কিছু Operations

set এর মধ্যে কোনো element আছে কিনা তা চেক করার জন্য আমরা in Operator টি ব্যবহার করতে পারি। থাকলে true return করবে আর না থাকলে false রিটার্ন করবে।

# initialize my\_set

my\_set = set("apple")

# check if 'a' is present

# Output: True

print('a' in my\_set)

# check if 'p' is present

# Output: False

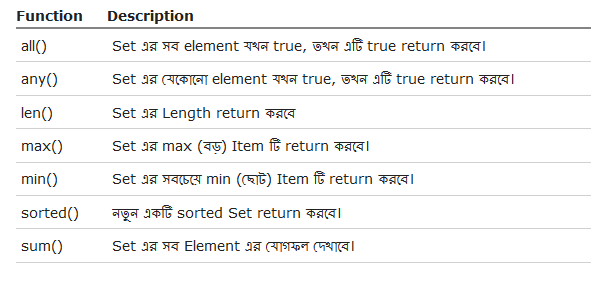
print('p' not in my\_set)

## Iterating Through a Set

পাইথনের for loop দিয়ে আপনি Set এর item গুলোকে iterate করতে পারেন ।

>>> for letter in set(“apple”):  
… print(letter)  
…  
a  
p  
e  
l

## Set এর কিছু Built-in Function List



>>> mysets={4,1,9,3,2,6,7,8}

>>> len(mysets)

8

>>> max(mysets)

9

>>> min(mysets)

1

>>> sum(mysets)

40

>>> sorted(mysets)

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

>>> any(mysets)

True

>>> all(mysets)

True