Python Git-book

Chapter

পাইথন ইন্ট্রোডাকশন

> 1.0: পাইথন ইন্ট্রোডাকশন

Data Types এবং Variables

আজকে আমরা পাইথন এর কিছু ব্যাসিক ডাটা টাইপ যেমন ইন্টিজার , ফ্লোট , স্ট্রিং , বুলিয়ান সম্পর্কে জানবো । সবথেকে মজার ব্যাপার হচ্ছে পাইথন এ ভ্যারিয়েবল এর নাম লিখে তারপর ভ্যালু দিলেই পাইথন বুঝে নেয় যে এটা কোন টাইপ ভ্যারিয়েবল। নিচের কোড টা খেয়াল করো

```
age = 12
name = "karim"
is_single = True
interest = 12.50

print(age, type(age))
print(name, type(name))
print(is_single, type(is_single))
print(interest, type(interest))
```

output :

```
12 <class 'int'>
karim <class 'str'>
True <class 'bool'>
12.5 <class 'float'>
```

Commenting in Python

মেইনলি দুই টাইপ এর কমেন্ট আছে
১. Single line Comment
২. Multi line Comment/ Doc String

```
# print(age, type(age)) A single line Comment. Press Ctrl+ /

"""
Multi Line Comment Press Alt + Shift + A
print(name, type(name))
print(is_single, type(is_single))
print(interest, type(interest))

"""
```

F String :
F স্ট্রিং দিয়ে তুমি চাইলে মাল্টিপল ভ্যারিয়েবল কে সুন্দর ভাবে ফরম্যাট করে প্রিন্ট করতে পারবে । তবে সেক্ষেত্রে ভ্যারিয়েবল গুলা সেকেন্ড ব্রাকেট এর মধ্যে রাখতে হবে

```
name = 'Tushar'
age = 23
print(f"Hello, My name is {name} and I'm {age} years old.")
```

output :

```
Hello, My name is Tushar and I'm 23 years old.
```

> ১.৪: ব্যাসিক ইনপুট আউটপুট এবং টাইপকাস্টিং

Basic Input and Output and Typecasting

পাইথন এ ইনপুট নেওয়া বাকি সকল প্রোগ্রামিং ল্যাঙ্গুয়েজ থেকে অনেক বেশি সহজ । জাস্ট ইংলিশ কীওয়ার্ড input এই নামের function ইউজ করলেই ইনপুট নেওয়া যায় । তবে মাথায় রাখতে হবে যে ইনপুট ফাংশন সবসময় স্ট্রিং ফরমেট এ ডাটা স্টোর করে । তাই একটা ইন্টেজার ইনপুট নিলেও সেটা স্ট্রিং হয়ে যাবে । সেক্ষেত্রে টাইপকাস্টিং করে সেটাকে ইন্টেজার এ কনভার্ট করে নিতে হবে

```
name = input("enter your name : ")
print(name)
```

```
enter your name : rahim
rahim
```

```
first_money = input("Enter first amount : ")
second_money = input("Enter second amount : ")
print(first_money + second_money)
```

output :

```
Enter first amount : 100
Enter second amount : 200
100200
```

after typecasting :

```
first_money = input("Enter first amount : ")
second_money = input("Enter second amount : ")
print(int(first_money) + int(second_money))
```

```
Enter first amount : 100
Enter second amount : 200
300
```

ব্যাসিক অপারেটরস

অন্য প্রোগ্রামিং ল্যাঙ্গুয়েজের মতো পাইথন এও নরমাল যোগ , বিয়োগ, গুন্ , ভাগ , কম্পারিসন করা ইত্যাদি করা যায়

Basic Operators in Python :

```
print(10+3)
print(10-3)
print(10*3)
print(10 % 3) # vagsesh
print(10/3)  # decimal vagfol
print(10//3)  # integer vagfol
```

output :

```
13
7
30
1
3.3333333333333335
3
```

Comparison Operator in Python :

1. Equal to (==): Checks if the values of two operands are equal.

```
x = 5
y = 5
print(x == y)  # Output: True

x = 5
y = 10
print(x == y)  # Output: False
```

2. Not equal to (!=): Checks if the values of two operands are not equal.

```
x = 5
y = 5
print(x ≠ y)  # Output: False

x = 5
y = 10
print(x ≠ y)  # Output: True
```

3. Greater than (>): Checks if the value of the left operand is greater than the value of the right operand.

```
x = 10
y = 5
print(x > y)  # Output: True

x = 5
y = 10
print(x > y)  # Output: False
```

4. Less than (<): Checks if the value of the left operand is less than the value of the right operand.

```
x = 10
y = 5
print(x < y)  # Output: False

x = 5
y = 10
print(x < y)  # Output: True
```

5. Greater than or equal to (>=): Checks if the value of the left operand is greater than or equal to the value of the right operand.

```
x = 10
y = 5
print(x ⩾ y)  # Output: True

x = 5
y = 5
print(x ⩾ y)  # Output: True
```

6. Less than or equal to (<=): Checks if the value of the left operand is less than or equal to the value of the right operand.

```
x = 10
y = 5
print(x ⩽ y)  # Output: False

x = 5
y = 5
print(x ⩽ y)  # Output: True
```

> ১.৬: কন্ডিশনাল স্টেটমেন্ট

Conditional Statement

অন্য প্রোগ্রামিং ল্যাঙ্গুয়েজ থেকে পাইথন এ কন্ডিশনাল স্টেটমেন্ট এ কিছুটা চেঞ্জ আছে । if কীওয়ার্ড যেমন ছিল তেমন ই আছে , else if টা পাইথন এ elif নামে ইউজ হয় আর else যেমন ছিল তেমন ই আছে । আরেকটা বিষয় হচ্ছে ব্রাকেট ইউজ করা হয় না কোনো কন্ডিশনাল ব্লক বুঝানোর জন্যে বরং ইউজ করা হয় ইনডেন্টেশন

Some examples of if, if-elif, if-elif-else, and nested if-elif statements:

```
if statement:
```

```
# Example: Check if a number is positive

num = 5
if num > 0:
    print("The number is positive.")
```

Output:

```
The number is positive.
```

2.

If-elif statement:

```
# Example: Check if a number is positive, negative, or zero

num = 5
if num > 0:
    print("The number is positive.")
elif num < 0:
    print("The number is negative.")
```

Output:

```
The number is positive.
```

3.

If-elif-else statement:

```
# Example: Check if a number is positive, negative, or zero

num = 5
if num > 0:
    print("The number is positive.")
elif num < 0:
    print("The number is negative.")
else:
    print("The number is zero.")
```

Output:

```
The number is positive.
```

4.

```
# Example: Check if a number is positive, negative, or zero,
with nested if-elif statements

num = 5
if num ⩾ 0:
    if num == 0:
        print("The number is zero.")
    else:
        print("The number is positive.")
else:
    print("The number is negative.")
```

Output:

```
The number is positive.
```

> ১.৭: while লুপ

While Loop

1.

While Loop

```
# Example: Print numbers from 1 to 5 using a while loop

num = 1
while num ⩽ 5:
    print(num)
    num += 1
```

Output:

```
1
2
3
4
5
```

2.

While Loop with if-elif Statement

```
# Example: Print numbers from 1 to 10 and classify them as
even or odd using a while loop with if-elif statement

num = 1
while num ⩽ 10:
    if num % 2 == 0:
        print(num, "is even")
    else:
        print(num, "is odd")
    num += 1
```

Output:

```
1 is odd
2 is even
3 is odd
4 is even
5 is odd
6 is even
7 is odd
8 is even
9 is odd
10 is even
```

3

While Loop with continue Statement

```
# Example: Print numbers from 1 to 5, skipping even numbers
using a while loop with continue statement

num = 1
while num ⩽ 5:
    if num % 2 == 0:
        num += 1
        continue
    print(num)
    num += 1
```

Output:

```
1
3
5
```

4.

While Loop with Break statement

```
# Example: Print numbers from 1 onwards until reaching a
number greater than 5 using a while loop with break statement

num = 1
while True:
    print(num)
    if num > 5:
        break
    num += 1
```

Output:

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

> ১.৮: for লুপ

For Loop

সি , সি ++ এ ফর লুপে যেমন ইনিশিয়ালাইজেশন , কন্ডিশন , ইনক্রিমেন্ট , ডিক্রিমেন্ট ইউজ করা হয় , পাইথন এ এইগুলার প্রয়োজন নাই আসলে । এখানে range নামে একটা মেথড ইউজ করি আমরা সেখানে initialization , ইনক্রিমেন্ট , ডিক্রিমেন্ট দেওয়া যায় আর অটোমেটিক কন্ডিশন apply হয়ে যায় ।

```
range([start], stop[, step])
```

রেঞ্জ মেথড এর কিছু এক্সাম্পল খেয়াল করি

```
# Example 1: range with only stop argument
for i in range(5):
    print(i)  # Output: 0, 1, 2, 3, 4

# Example 2: range with start and stop arguments
for i in range(2, 5):
    print(i)  # Output: 2, 3, 4

# Example 3: range with start, stop, and step arguments
for i in range(1, 10, 2):
    print(i)  # Output: 1, 3, 5, 7, 9
```

এখন কিছু ফর লুপের এক্সাম্পল খেয়াল করি

1.

For Loop

```
# Example: Print numbers from 1 to 5 using a for loop

for num in range(1, 6):
    print(num)
```

Output:

```
1
2
3
4
5
```

2.

For Loop with if-elif statement

```
# Example: Print numbers from 1 to 10 and classify them as
even or odd using a for loop with if-elif statement

for num in range(1, 11):
    if num % 2 == 0:
        print(num, "is even")
    else:
        print(num, "is odd")
```

Output:

```
1 is odd
2 is even
3 is odd
4 is even
5 is odd
6 is even
7 is odd
8 is even
9 is odd
10 is even
```

3.

For Loop with continue statement

```
# Example: Print numbers from 1 to 5, skipping even numbers
using a for loop with continue statement

for num in range(1, 6):
    if num % 2 == 0:
        continue
    print(num)
```

Output:

```
1
3
5
```

4.

For Loop with break statement

```
# Example: Print numbers from 1 onwards until reaching a
number greater than 5 using a for loop with break statement

for num in range(1, 10):
    print(num)
    if num > 5:
        break
```

Output:

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

5.

For Loop with string

```
# Example: Iterate over each character in a string

word = "Python"
for char in word:
    print(char)
```

Output:

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

6.

For Loop with list

```
# Example: Iterate over each element in a list

numbers = [1, 2, 3, 4, 5]
for num in numbers:
    print(num)
```

Output:

```
1
2
3
4
5
```

ব্যাসিক পাইথন

> ২.১ঃ ইন্ট্রোডাকশন টু ফাংশন ইন পাইথন

Introduction To Function

পাইথন এ ফাংশন লিখার জন্য আমরা def কিওয়ার্ড ব্যবহার করবো। নিচে একটা পাইথন ফাংশন উদাহরন দেওয়া হলো।

```
def sum(a, b):
    return a + b

result = sum(5, 4)
print(result)

# output: 9
```

এখানে sum হচ্ছে ফাংশন নেম a, b হচ্ছে দুইটা প্যারামিটার এবং এদের যোগফল রিটার্ন স্টেটমেন্ট এর মাধ্যমে রিটার্ন করা হয়েছে। আমরা যদি কোনো ফাংশন থেকে কোনোকিছু রিটার্ন না করি তাহলে বাই ডিফল্ট সেটা None রিটার্ন করে ।

! মনে রাখতে হবে পাইথন এর ক্ষেত্রে ইন্ডেন্টেশন অনেক গুরুত্বপুর্ন।

> ২-২ঃ ডিফল্ট প্যারামিটার এবং args ইন পাইথন

Default parameter & args

পাইথনে প্যারামিটার হিসেবে আমরা ডিফল্ট প্যারামিটার এড করতে পারি যেমন-

```
def sum(a, b, c = 0):
    return a + b + c

result = sum(5, 4)       #output 5 + 4 + 0 = 9
result2 = sum(5, 4, 3)   #output 5 + 4 + 3 = 12
```

এখানে c = 0 লিখার মাধ্যমে আমরা একটা ডিফল্ট প্যারামিটার ডিক্লেয়ার করছি c = 0 এর মানে আমরা যদি এই প্যারামিটার এ কোনো ভ্যালু পাস করি তাহলে c এর মান পাস করা ভ্যালুটা হবে কিন্তু যদি আমরা পাস না করি তাহলে c এর মান 0 থাকবে

এখন যদি কখনো আমাদের এরকম প্রয়োজন পরে যে আমরা কতগুলো প্যারামিটার পাস করব সেটা আমরা জানিনা সেক্ষেত্রে আমরা args ব্যবহার করতে পারি । args ব্যবহারের জন্য প্যারামিটার এর নামের পুর্বে * এড করে দিতে হয়।

```
def sum(*args):
    sum = 0
    for num in args:
        sum = sum + num

result = sum(5, 4)     #output 5 + 4 = 9// Some code
```

args ব্যবহারে ভ্যালুগুল একটা টাপল (টাপল সম্পর্কে আমরা পরবর্তিতে বিস্তারিত জানব) হিসেবে আসে। যেটার উপরে লুপ চালিয়ে আমরা ভ্যালুগুলোকে এক্সেস করতে পারি।

> ২-৩ঃ kwargs এবং ফাংশন থেকে মাল্টিপল রিটার্ন

kwargs & multiple function value return

args প্যারামিটার এর ক্ষেত্রে আমাদেরকে ভ্যালুগুলো সিরিয়াল অনুযায়ী দিতে হতো। আমরা চাইলে প্যারামিটার গুলোকে কি ভ্যালু পেয়ার আকারেও দিতে পারি যেটাকে kargs বলে যেমনঃ-

```
def func( **kwargs):
    print(f"apple: {kwargs['apple']}")
    for key, value in kwargs.items():
        print(f'{key} : {value}')

result = func(apple = 5, orange= 4)
```

যখন আমরা প্যারামিটার kwargs হিসেবে পাস করি তখন সেটা একটা ডিকশনারি (ডিকশনারির ব্যাপারে আমরা পরবর্তিতে জানব) আকারে থাকে এটা থেকে ভ্যালু এক্সেস করতে হলে kwargs['key_name'] এইভাবে এক্সেস করতে পারি। আমরা চাইলে উপরের উদাহরণের মতো কী, ভ্যালু আকারেও ভ্যালু এক্সেস করতে পারি।

আমরা যদি কখনো পাইথন ফাংশন থেকে মাল্টিপল ভ্যালু রিটার্ন করতে চাই তাহলে সেটাকে আমরা কমা সেপারেটেড ভ্যালু হিসেবে রিটার্ন করতে পারি সেক্ষেত্রে এটা টাপল আকারে রিটার্ন করবে এবং আমরা চাইলে লিস্ট বা ডিকশানারি আকারেও ভ্যালুগুলোকে রিটার্ন করতে পারি

```
def func( a, b, c):
    return a, b, c           # return as tuple (a, b, c)
    return [a, b, c]         # return as list [a, b, c]
    return {'a': a, 'b': b}  # return as dictionary
```

> ২-৪ঃ লোকাল স্কোপ এবং গ্লোবাল স্কোপ

Local Scope & Global Scope

Global Scope

আমরা যদি কোনো একটা ভেরিয়েবল কে ফাংশন এর বাহিরে ডিক্লেয়ার করি তাহলে সেটাকে বলে গ্লোবাল ভেরিয়েবল গ্লোবাল ভেরিয়েবল কে আমরা প্রোগ্রাম এর সব যায়গা থেকে এক্সেস করতে পারি যেটাকে বলে গ্লোবাল স্কোপ তবে যদি আমরা স্পেসিফিক ভাবে গ্লোবাল ভেরিয়েবল এর উপর কোনো চেঞ্জ করতে চায় সেক্ষেত্রে global কীওয়ার্ড ব্যবহার ক্রয়তে হবে

```
balance = 500

def func():
    print(balance) # Accessible

func()
print(balance) # Accessible

def chk():
    balance = balance - 5
    # ইরর ত্রো করবে কারন balance টাকে প্রোগ্রাম নতুন ভেরিয়েবল মনে করবে তাই এটার সল্যুশন
    # এইভাবে লিখা

def chk():
    global balance
    balance = balance - 5

chk()
```

Local Scope

কোনো একটা ভেরিয়েবল কে যদি আমরা কোন একটা ফাংশন এর ভিতরে ডিক্লেয়ার করি তাহলে সেটা লোকাল ভেরিয়েবল এবং এই লোকাল ভেরিয়েবল কে শুধুমাত্র সেই ফাংশন এর মধ্যে থেকেই এক্সেস করা সম্ভব অন্য কোথায় থেকে এটাকে এক্সেস করা যাবেনা এইযে ভেরিয়েবল টাকে শুধু লোকালি এক্সেস করা যাচ্ছে এটাকে বলে লোকাল স্কোপ

```
def func():
    balance = 500
    print(balance)   # Accessible

print(balance)       # Not Accessible
```

Built-in Function & Import Module

Built In Function

পাইথনে অনেকগুলো বিল্ট ইন ফাংশন রয়েছে যেগুলো দিয়ে অনেকগুলো কাজকে সহজেই করে ফেলা যায় এসকল বিল্ট ইন ফাংশন এর লিস্ট এবং ডিটেইলস পেয়ে যাবে এখানে।

এর মধ্যে কিছু বিল্ট ইন ফাংশন এর বর্ননা নিচে দেওয়া হলো--

max()

এটা যেকোনো আইটেরেবল (লিস্ট, টাপল, স্ট্রিং) থেকে ম্যাক্সিমাম ইলেমেন্ট টাকে বের করে আনে

```
highest = max([3,4,5,2])
print(highest)
# output: 5
```

min()

এটা যেকোনো আইটেরেবল (লিস্ট, টাপল, স্ট্রিং) থেকে মিনিমাম ইলেমেন্ট টাকে বের করে আনে

```
highest = min([3,4,5,2])
print(highest)
# output: 2
```

Functioon	Description
abs()	Return the absolute value of a number.
all()	Return True if all elements of the iterable are true.
any()	Return True if any element of the iterable is true.
bin()	Convert an integer number to a binary string.
bool()	Return a Boolean value.
dict()	Create a new dictionary.
dir()	Return the list of names in the current local scope.
enumerate()	Return an enumerate object.

filter()	Construct an iterator from an iterable and returns true.
float()	Return a floating point number from a number or string.
getattr()	Return the value of the named attribute of object.
input()	This function takes an input and converts it into a string.
len()	Return the length (the number of items) of an object.
list()	Rather than being a function, list is a mutable sequence type.
map()	Return an iterator that applies function to every item of iterable.
max()	Return the largest item in an iterable.
min()	Return the smallest item in an iterable.
pow()	Return base to the power exp.
print()	Print objects to the text stream file.
repr()	Return a string containing a printable representation of an object.
reversed()	Return a reverse iterator.
round()	Return number rounded to ndigits precision after the decimal point.
set()	Return a new set object.
slice()	Return a sliced object representing a set of indices.
sorted()	Return a new sorted list from the items in iterable.
sum()	Sums start and the items of an iterable.
super()	Return a proxy object that delegates method calls to a parent or sibling.
tuple()	Rather than being a function, is actually an immutable sequence type.
type()	Return the type of an object.

import()	This function is invoked by the import statement.
Open()	Open file and return a corresponding file object.
Object()	Return a new featureless object.

> !! এরকম আরো অনেক ফাংশন আছে তোমার কাজ হচ্ছে ফাংশনগুলো পড়ে রান করে দেখা।

Import

মনে করে আমাদের functions.py ফাইল এর মধ্যে একটা ফাংশন আছে sum() নামে এবং আমরা এই ফাইলটিকে আরেকটি ফাইল modules.py থেকে এক্সেস করতে চাচ্ছি তাহলে আমাদেরকে modules.py ফাইলের মধ্যে sum ফাংশনটাকে ইম্পোর্ট করতে হবে এইভাবে

```
# functions.py
def sum(a, b):
    return a+b

# modules.py
from functions import sum

result = sum(5, 6)
```

!! এখানে একটা জিনিস মনে রাখতে হবে এইভাবে ইম্পোর্ট এর জন্য functions.py এবং modules.py ২ টা ফাইল ই একই ফোল্ডার এর মধ্যে থাকতে হবে যদি অন্য কোনো ফোল্ডার এর মধ্যে থাকে তাহলে সেটার পাথ সহ লিখতে হবে যদি ফাইল ফোল্ডার এইভাবে থাকে

folder

-functions.py

modules.py

তাহলে **modules.py** এ ইম্পোর্ট পাথ হবে এরকম

from folder.modules import sum

যদি আমরা একই ফাইল থেকে মাল্টিপল ফাইল কে ইম্পোর্ট করতে চাই সেটা এইভাবে লিখা হয়

```
from functions import sum, multiply
```

এবং যদি আমরা functions.py ফাইল এর সবগুলো ফাংশন একসাথে ইম্পোর্ট করতে চাই তাহলে এইভাবে লিখতে হবে

```
from functions import *
```

> ২.৬ঃ লিস্ট, লিস্ট ইনডেক্স ফাংশন এবং স্লাইস ইন পাইথন

List, List index & slice

লিস্ট এ আমরা array মতো করে চিন্তা করতে পারি তবে C/C++ এর এরের তুলনায় পাইথনের লিস্ট এ কিছু ভিন্নতা আছে।

পাইথন এ এইভাবে লিস্ট কে ডিফাইন করা হয়

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

এবং লিস্টে ২ ধরনের ইন্ডেক্সিং রয়েছে শুরু থেকে শেষ পর্যন্ত ইন্ডেক্সিং হচ্ছে এরকম 0, 1, 2 ... n-1 আবার উল্টা দিক থেকেও এটার ইন্ডেক্সিং সম্ভব এই ইনডেক্স টা এইভাবে হয় -1, -2, -3 ... -n

এই ইনডেক্সগুলো ইউস করে এইভাবে ইলেমেন্ট গুলোকে এক্সেস করতে পারি

```
numbers[0]  # এটার ভ্যালু ১
numbers[-1] # এটার ভ্যালু ৫
```

কোনো একটা লিস্টকে আমরা এইভাবে স্লাইস করে ফেলতে পারি

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

এটার সিন্ট্যাক্সটা এরকম list[start : end] যেখানে স্টার্ট থেকে শুরু করে end-1 পর্যন্ত ভ্যালু নিয়ে আরেকটা লিস্ট এটা আমাদের রিটার্ন করবে

আরেকটা সিনট্যাক্স হচ্ছে list[start : end : step] এখানে আমরা step ডিক্লেয়ার করে দিতে পারি নিচের উদাহরণটি দেখলে এটা ক্লিয়ার হয়ে যাবে

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8]
print(numbers[0:8:2])
#output: [1, 3, 5, 7]
```

আমরা এই স্টেপ এর টেকনিকটাকে উল্টাভাবে প্রিন্ট এর জন্যেও ব্যবহার করতে পারি

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8]
print(numbers[8:2:-1])
#output: [8, 7, 6, 5, 4]
```

!! এখন তোমাদের কাজ হচ্ছে ইনডেক্স গুলো চেঞ্জ করে করে আউটপুট পর্যবেক্ষন করে বোঝার চেষ্টা করা কি ঘটছে।

আমরা এখানে স্টার্ট বা এন্ড ইনডেক্স স্পেসিফিকভাবে ডিফাইন না করে দিলে এটা বাই ডিফল্ট স্টার্ট বা ইন্ড ইনডেক্স নিয়ে নেয় যেমন এটা ৩ নাম্বার ইনডেক্স থেকে শেষ পর্যন্ত প্রিন্ট করছে

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

একইভাবে এটা শুরু থেকে ৫-১ ইনডেক্স পর্যন প্রিন্ট করবে

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

তেমনি কোনো কিছু ম্যানশন না করে numbers[:] এইভাবে লিখলে পুরো লিস্ট টাই চলে আসবে

তেমনি আমরা যদি এইভাবে লিখি তাহলে পুরো লিস্ট টাই রিভার্স হয়ে যাবে numbers[::-1] এখানে আমরা start, end কোনটাই দেয়নি শুধু স্টেপ ডিক্লেয়ার করছি।

এরকম আরো মজার কিছু জিনিস জানতে এই **https://docs.python.org/3/tutorial/datastructures.html** থেকে সবকিছু দেখে ফেলো।

List Method

পাইথন এ অনেকগুলো লিস্ট মেথড আছে যেমন --

append()

এটা লিস্ট এর শেষে কোনো ভ্যালুকে ইন্সার্ট করে

```
numbers = [1, 2, 3, 4, 5]
numbers.append(6)
print(numbers)
# output: [1, 2, 3, 4, 5, 6]
```

insert()

কোনো নির্দিষ্ট ইনডেক্স এ যদি ইন্সার্ট করতে চাই তাহলে এই ফাংশনটি ইউস করা হয়, যেমন যদি আমরা ২ নাম্বার ইনডেক্স এ ১০০ ইন্সার্ট করতে চাই তাহলে

```
numbers = [1, 2, 3, 4, 5]
numbers.insert(2, 100)
print(numbers)
# output: [1, 2, 100, 3, 4, 5]
```

remove()

কোনো একটা ইলেমেন্ট পাস করলে সেটা লিস্ট থেকে রিমুভ করে দিবে যেমন

```
numbers = [1, 2, 3, 4, 5]
numbers.remove(3)
print(numbers)
# output: [1, 2, 4, 5]
```

রিমুভ এর জন্য ভ্যালু টা অবশ্যই লিস্ট এ থাকতে হবে

pop()

লিস্টের কোনো ইনডেক্স এর নির্দিষ্ট ইনডেক্স এর ভ্যালুকে পপ করা যায় **pop(i)** এইভাবে যদি কোনো ইনডেক্স না দেওয়া হয়ে তাহলে **list.pop()** লাস্ট এর ইলেমেন্টটাকে রিমুভ করে দিবে

`index()`
এই ফাংশনে কোনো একটা ভ্যালু দিলে সেটার ইনডেক্স রিটার্ন করে numbers.index(5) ৫ ভ্যালুটা যেই ইনডেক্স এ আছে সেটা রিটার্ন করবে

এরকম আরো অনেক মেথড আছে যেগুলো তুমি এই **https://www.pythoncheatsheet.org/cheatsheet/lists-and-tuples#the-index-method** থেকে দেখে নিতে পারো

> ২-৮ঃ লিস্ট কম্প্রিহেনশন

List Comprehension

লিস্ট কম্প্রিহেনশন এর মাধ্যমে আমরা একটা লাইনের মধ্যে লুপ এবং কন্ডিশন দিয়ে লিস্ট জেনেরেট করতে পারি যেমন-

```
numbers = [45, 87, 96, 65, 43, 90, 85, 14, 26, 61, 70]
odds = []
for num in numbers:
    if num % 2 == 1 and num % 5 == 0:
        odds.append(num)
```

উপরের কোড এর কাজটাকে আমরা লিস্ট কম্প্রিহেনশন দিয়ে এইভাবে লিখতে পারি

```
odd_numbers = [num for num in numbers if num % 2 == 1 if num % 5 == 0]
```

এখানে num সুরুতে রাখা হয়েছে যেটা লুপ থেকে আশা ভ্যালুগুলোকে লিস্ট এ রাখবে এবং এর জন্য কন্ডিশন এড করা হয়েছে প্রতিটা কন্ডিশন আলাদা আলাদা if দিয়ে লিখতে হয়

লিস্ট কম্প্রিহেনশন এর মাধ্যমে নেস্টেড লুপ ও দেখা যায় যেমন-

```
players = ['sakib', 'musfik', 'liton']
ages = [38, 37, 32]

age_comb = []
for player in players:
    for age in ages:
        age_comb.append([player, age])

print(age_comb)
```

এই কোডটিকে আমরা এইভাবে লিখতে পারি

```
players = ['sakib', 'musfik', 'liton']
ages = [38, 37, 32]

age_comb = [[player, age] for player in players for age in
ages]
print(age_comb)
```

ব্যাসিক পাইথন-2

> ৩.১ঃ স্ট্রিং ও স্ট্রিং মেথডস

String & String Method

এই মডিউলে আমরা পাইথন স্ট্রিং এবং কিছু স্ট্রিং মেথড নিয়ে আলোচনা করব।

শুরুতে মডিউল-৩ এর কোডগুলো রাখার জন্য Module-3 নামে একটি নতুন ফোল্ডার ক্রিয়েট করে VS Code এ ওপেন করে ফেলি।string_methods.py নামে একটি ফাইলও তৈরি করে ফেলি।

!! স্ট্রিং হলো ক্যারেক্টারের সিকুয়েন্স। ক্যারেকটারের সিকুয়েন্স কে সিংগেল ('...') অথবা ডাবল কোটেশনের ("...") মধ্যে রাখলে সেটি স্ট্রিং হিসেবে বিবেচিত হয়।

কথা না বাড়িয়ে একটি স্ট্রিং ডিক্লেয়ার করে ফেলি-

```
myString= 'I am Phitron'
```

উপড়ের স্ট্রিং এ 'P' এর ইনডেক্স কত?

আমি জানি, অবশ্যই আপনাদের উত্তর হবে 5, উত্তর অবশ্যই সঠিক। কিন্ত 'P' এর আরেকটি ইনডেক্স রয়েছে, সেটি হলো -7 (minus seven).

Sequence	I		a	m		P	h	I	t	r	O	N
Index (forward)	0	1	2	3	4	5	6	7	8	9	10	11
Index (Backward)	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

অর্থ্যাৎ, n সাইজের পাইথন স্ট্রিং এর n তম ক্যারেকটারের ইনডেক্স হবে -১ , তার আগের ক্যারেকটারের -২ ... এভাবে চলতে থাকবে।

স্ট্রিং কে খুবে সহজে slice করা যায়, যেমন 'I am Phitron' স্ট্রিং থেকে যদি আমরা **`'Phitron'`** অংশটুকু পেতে চাই তাহলে **`myString[5:12]`** লিখলেই পেয়ে যাব।

এই কাজটি অবশ্য Backward বা নেগেটিভ ইনডেক্স ব্যবহার করেও করে দেখতে পারেন।

!! কিন্তু একটি কাজ কোনো ভাবেই করতে পারবে না , সেটি হলো স্ট্রিং এর কোনো ইলিমেন্ট চেঞ্জ করা। যেমন , `myString[5]='T'` **লিখে রান করলে ইরোর খাবেন**

বেশ কিছু ইউসফুল স্ট্রিং মেথড সম্পর্কে জানতে এই আর্টিকেল টি পড়ে ফেলতে পারেন-

- **https://www.programiz.com/python-programming/methods/string**
- **https://www.pythoncheatsheet.org/cheatsheet/string-formatting**

> ৩.২ঃ টাপল ও টাপল মেথডস

Tuple & Tuple Method

আগের মডিউলে আমরা লিস্ট সম্পর্কে জেনেছি। এই মডিউলে জানব লিস্ট এর কাছাকাছি একটী ডেটা স্ট্রাকাচার, টাপল সম্পর্কে।

টাপল হলো কমা দিয়ে সেপারেট করা অবজেক্টের লিস্ট। তার মানে টাপলের ইলিমেন্ট হিসেবে আমরা নাম্বার,স্ট্রিং,লিস্ট ... ইত্যাদি যে কোনো কিছু রাখতে পারি। বোঝাই যাচ্ছে টাপলের সাথে লিস্টের বেশ সাদৃশ্য রয়েছে, যেমন ইনডেক্সিং, লুপিং অপারেশন গুলো লিস্টের মতোই। একটী টাপল ডিক্লেয়ার করে ফেলি-

```
myTuple=['Hello',[1,2,3],17]
```

!! লিস্টের সাথে টাপলের সবচেয়ে বড় অমিল হলো লিস্ট মিউটেবল (পরিবর্তনযোগ্য) কিন্তু টাপল ইমিউটেবল (অপরিবর্তনযোগ্য)। এর জন্য নিচের কোড স্নিপেট টি রান করলে ইরোর আসবে-

```
myTuple=['Hello',[1,2,3],17]

myTuple=[0]='World'
```

কিন্তু, মজার ব্যাপার হলো টাপলের কোনো ইনডেক্সে যদি মিউটেবল ইলিমেন্ট থাকে তাকে কিন্তু আমরা পরিবর্তন করতে পারব। যেমন-

```
myTuple=[1][0]=5

print(myTuple[1])        # output: [5, 2, 3]
```

কিছু Tuple মেথড সম্পর্কে জানতে এই আর্টিকেল টি পড়ে ফেলতে পারেন-
https://www.geeksforgeeks.org/python-tuple-methods/

> ৩.৩ঃ সেট ও সেট মেথডস

Set & Set Method

এই মডিউলে আমরা পাইথনের সেট ডেটা স্ট্রাকচার সম্পর্কে জানব।

আগের মডিউলে দেখেছি, লিস্ট ডিক্লেয়ার করতে হয় [ ] ব্যবহার করে, টাপল ডিক্লেয়ার করা হয় ( ) ব্যবহার করে, এই মডিউলে দেখব সেট ডিক্লেয়ার করা হয় { } ব্যবহার করে-

```
mySet={3,5,6,7,8,2,2,3}

print(mySet)

# output: {2, 3, 5, 6, 7, 8}
```

!! mySet সেটে ২ দুইবার, ৩ দুইবার ছিল। কিন্ত আউটপুটে একবার করে আসল !! আরেকটি ব্যাপার , ইলিমেন্ট গুলো সর্ট হয়ে আসল !!

✓ তার মানে, সেট হলো ইউনিক ডেটার কালেকশন, অর্থ্যাৎ , সেটে কোনো ডুপ্লিকেট ভ্যালু থাকবে না ।

সেট মিউটেবল, কিন্ত এসাইনমেন্ট অপারেশন এক্সেপ্ট করে না, add( ) , remove( ) ফাংশন ব্যবহার করে ডেটা এ্যাড, ডিলিট করা যায়। সেই সাথে কিছু সেট অপারেশন যেমন |(Union), &(Intersection) ইত্যাদি ব্যবহার করা যায়-

```
mySet={3,5,6}
yourSet={6,7,8}

ourSet= mySet & yourSet

print(ourSet)         # output: {3}
```

আরো কিছু সেট মেথড সম্পর্কে জানতে এই আর্টিকেল টি পড়ে ফেলতে পারেন-
https://www.freecodecamp.org/news/python-set-how-to-create-sets-in-python/
https://www.pythoncheatsheet.org/cheatsheet/sets

> ৩-৪ঃ ডিকশনারি ও ডিকশনারি মেথডস

Dictionary & It's methods

এই মডিউলে আমরা পাইথন ডিকশনারি সম্পর্কে জানব।
ডিকশনারি হলো **key** এবং **value** জোড়া হিসেবে ডেটা ম্যানেজ করার উপায়।

```
myDictionary={‘name':'Ratin','age':27,'courses':
[‘Database','Python','Django']}
```

Dictionary মিউটেবল। ডিকশনারি থেকে ডেটা এক্সেস করার উপায়-

```
print(myDictionary[‘name’])     # output: Ratin
```

ডিকশনারি কে নিচের মত করে ইটারেট করা যাবে-

```
myDictionary={'name':'Ratin','age':27, 'courses':
['Database','Python','Django']}

for key,val in myDictionary.items():
    print(key,val)
```

Output:

```
name Ratin
age 27
courses ['Database', 'Python', 'Django']
```

অবশ্য আমরা চাইলে ইটারেট করার সময় key এভয়েড করতে পারি -

```
myDictionary = {"name": "Ratin", "age": 27, "courses":
["Database", "Python", "Django"]}

for _, val in myDictionary.items():
    print(val)
```

```
# Output:

Ratin
27
['Database', 'Python', 'Django']
```

ডিকশনারিতে থাকা সবগুলো কী বা ভ্যালু আমরা পেতে চাইলে keys() , values() ফাংশন ব্যবহার করতে পারি।

```
myDictionary = {"name": "Ratin", "age": 27, "courses":
["Database", "Python", "Django"]}

print(myDictionary.keys())
print(myDictionary.values())
```

```
# Output:

dict_keys(['name', 'age', 'courses'])
dict_values(['Ratin', 27, ['Database', 'Python', ‘Django']])
```

> ৩-৫ঃ বিল্ট-ইন মডিউলসঃ- Time, Math, Random ইত্যাদি

Time, Math, Random Built-in Modules

পাইথনে বেশ কিছু বিল্ট-ইন মডিউলস রয়েছে।

নিচের আর্টিকেল থেকে আমরা পাইথনের বিল্ট-ইন মডিউলস গুলো সম্পর্কে জানতে পারি- https://docs.python.org/3/py-modindex.htmlBuilt-in মডিউলের একটি উদাহরন দেখে ফেলা যাক।

র‍্যান্ডম সংখ্যা জেনারেট করতে আমরা random( ) ব্যবহার করতে পারি-

```
from random import *

print(random( ))
```

```
# output:

0.5014144055218996
0.7883877357162421
0.0408731603353123
```

দেখা যাচ্ছে , random() ফাংশনটি ০ থেকে ১ রেঞ্জে র‍্যান্ডম সংখ্যা রিটার্ণ করছে।

আমরা যদি Int র‍্যান্ডম সংখ্যা পেতে চাই তবে randint( ) ব্যবহার করতে পারি-

```
from random import *

print(randint(50,500))
```

```
# output:

54
157
344
236
```

এখানে, randint(50,500) ফাংশনটি ৫০ থেকে ৫০০ রেঞ্জের মধ্যে ইন্টিজার সংখ্যা রিটার্ন করেছে।

> ৩-৬: এক্সটার্নাল প্যাকেজ ইন্সটলেশন

External Package Installation

পাইথনে এক্সটার্নাল প্যাকেজ ইনস্টল করার জন্য আমাদেরকে pip ইউস করতে হবে।
√ আমরা যখন পাইথন ইনস্টল করেছিলাম, তখন pip ও ইনস্টল হয়ে যাবার কথা ।

PIP ইন্সটল আছে কি না চেক করার জন্য টার্মিনালে নিচের কমান্ড চালাই -
`pip –version`

দেখা যাচ্ছে, আমার ডিভাইসে পিপ ইনস্টল করা আছে।
এখন এই পিপ ব্যবহার কওরে আমরা বেশ কিছু এক্সটার্নাল প্যাকেজ ব্যবহার করতে পারব।
বেশ কিছু এক্সটার্নাল প্যাকেজের মধ্যে PyAutoGUI অন্যতম।
PyAutoGUI ইনস্টল করার জন্য টার্মিনালে নিচের কমান্ড রান করি-

```
pip install pyautogui
```

PyAutoGUI এর ব্যবহার বিস্তারিত দেখতে নিচের আর্টিকেল টি ফলো করতে পারেন-
https://pyautogui.readthedocs.io/en/latest/

> ৩.৭ঃ Try, Except, Finally ও ফাইল ম্যানেজমেন্ট

Try, Except, Finally and File management

```
try:
    # result = 45/0
    result = 45/5
except:
    print('error happened')
finally:
    print('finally here')
print('Done')
```

File Management

```
# .csv comma separated value
# .txt text file

# with open('message.txt', 'w') as file:
#     file.write('I love you, python!')

# with open('message.txt', 'a') as file:
#     file.write('I love you, python!')

with open('message.txt', 'r') as file:
    text = file.read()
    print(text)
```

> ৩.৮ঃ ল্যামডা ফাংশন

Lamda Function

In Python, a lambda function is a single-line, anonymous function, which can have any number of arguments, but it can only have one expression.

Resource
https://www.pythoncheatsheet.org/cheatsheet/functions#lambda-functions

Syntax

```
lambda arguments : expression
```

The expression is executed and the result is returned:

```
x = lambda a : a + 10
print(x(5))
```

ক্লাস এবং অবজেক্ট

> ৫.১ঃ ইন্ট্রোডাকশন টু সিম্পল ক্লাস

Introduction to simple class

অবজেক্ট ওরিয়েন্টেড প্রোগ্রামিং এ ক্লাস হচ্ছে একটা ইউজার ডিফাইনড ডেটা টাইপ অথবা কোনো একটা অবজেক্ট এর ব্লুপ্রিন্ট বলা যায়। একটা ক্লাস কে টেম্পলেট ডেফিনেশন বলা হয় যেখানে কোনো একটা অবজেক্ট এর জন্য ভেরিয়েবলস, মেথড ইত্যাদি ডিক্লেয়ার করা হয়।

উদাহরণস্বরূপ আমরা একটা সিম্পল পাইথন ক্লাস তৈরি করি

```
class Phone:
    price = 19000
    color = 'blue'
    brand = 'samsung'

myphone = Phone()
print(myphone.brand)    # output: samsung
```

এখানে class কিওয়ার্ড ব্যবহার করে আমরা Phone নামে একটা ক্লাস তৈরি করেছি এবং সেই ক্লাস এর একটি অবজেক্ট হচ্ছে myphone এই অবজেক্ট এর মাধ্যমে আমরা myphone এর প্রোপার্টি যেমন price, color, brand এক্সেস করতে পারি।

> ৫-২ঃ মেথড তৈরি এবং এর ব্যবহার

Create method & it’s use case

আমরা আগের মডিউল এ একটা সিম্পল ক্লাস ডিক্লেয়ার করেছিলাম যেখানে কিছু ক্লাস ভেরিয়েবল ডিক্লেয়ার করেছিলাম আমরা একটা ক্লাস এর মধ্যে ভেরিয়েবল ছাড়াও মেথড ও ডিক্লেয়ার করতে পারি একটা মেথড কোনো একটা ভেরিয়েবল এর বিহেবিয়ার কে নির্দেশ করে যেমন আমাদের **Phone** ক্লাস এর একটা বিহেবিয়ার হতে পারে **`Call`** তাহলে এই নামে আমরা একটা মেথড ডিক্লেয়ার করতে পারি এইভাবে

```
class Phone:
    price = 19000
    color = 'blue'
    brand = 'samsung'

    def call(self):
        print('Calling one person')

myphone = Phone()
myphone.call()      #output: Calling one person
```

এখানে **`call()`** নামে যেই ফাংশন এর মতো ডিক্লেয়ার করা হয়েছে ক্লাস এর মধ্যে এটাই মেথড। ফাংশন যখন কোনো একটা ক্লাস এর আন্ডারে ডিফাইন করা হয় তখন সেটাকে মেথড বলে এবং এই মেথডটি প্রতিটি অবজেক্ট এর জন্য স্বতন্ত্রভাবে কাজ করে যেমন **`myphone.call()`** এর মাধ্যমে **`myphone`** নামের অবজেক্ট এর **`call`** মেথডকে কল করা হয়েছে।

!! মনে রাখতে হবে পাইথনে মেথড ডিক্লেয়ার করার ক্ষেত্রে প্রথম প্যারামিটার হিসেবে `self` দিতে হবে

এবং আমরা চাইলে একটা ক্লাস এর মধ্যে মাল্টিপল মেথডও ক্রিয়েট করতে পারি এইভাবে

```
class Phone:
    price = 19000
    color = 'blue'
    brand = 'samsung'

    def call(self):
        print('Calling one person')

    def send_message(self, message):
        return f"Sending message: {message}"

myphone = Phone()
myphone.call() #output: Calling one person
print(myphone.send_message("Hello World"))

#output:Sending message: Hello World
```

একটা ফাংশন এর মতো করে আমরা মেথড এও প্যারামিটার নিতে পারি এবং মেথড থেকে কোনোকিছু রিটার্ন ও করতে পারি

> ৫.৩ঃ কন্সট্রাক্টর এবং __init__ ইন পাইথন

Constructor & __init__

এতক্ষণ পর্যন্ত আমরা যেই Phone ক্লাস নিয়ে কাজ করছিলাম সেখানে একটা সমস্যা ছিল আমরা ফোন ক্লাস এর যতগুলো অবজেক্ট ই ডিক্লেয়ার করি কিনা তাদের সবার প্রোপার্টির মান একই থাকছে কারন আমরা ক্লাস এর মধ্যে এই প্রোপার্টির মানগুলো সেট করে ফেলসি। কিন্তু ক্লাস তৈরির মেইন উদ্দেশ্য কিন্তু এটা ছিলোনা আমাদের মেইন উদ্দেশ্য হচ্ছে আমরা সেম ক্লাস দিয়ে ভিন্ন ভিন্ন ভ্যালুর অবজেক্ট তৈরি করব অর্থাৎ ক্লাসটাকে একটা টেম্পলেট হিসেবে ব্যবহার করব

ঠিক এই কাজটা করার জন্য কন্সট্রাক্টর ব্যবহার করা হয়। যখনই আমরা কোনো অবজেক্ট ডিক্লেয়ার করি তখন বাই ডিফল্ট সেই ক্লাস এর কন্সট্রাক্টর কল হয় এবং আমরা এই কন্সট্রাক্টর এর মাধ্যমে কোনো একটা অবজেক্ট এর প্রোপার্টি ভ্যালু এর মান সেট করতে পারি যেমন

```
class Phone:
    def __init__(self, brand, price):
        self.brand = brand
        self.price = price

    def call(self):
        pass

samsung = Phone("Samsung", "90000")
iphone = Phone('Apple', '150000')
print(samsung.brand)    #output: Samsung
print(iphone.brand)     #output: Apple
```

এখনে লক্ষ করি **samsung** এবং **iphone** নামে আমরা ২ টা অবজেক্ট ক্রিয়েট করছি **Phone** ক্লাস ইউস করে এবং **Phone("Samsung", "90000")** এর মাধ্যমে আমরা ফোন ক্লাস এর কন্সট্রাক্টর কে কল করেছি যেটা কিনা পাইথন এ **def __init__()** এইভাবে ডিক্লেয়ার করা হয়। এই কন্সট্রাক্টর এর মধ্যে ভালুগুলোকে পাস করার মাধ্যমে **brand** এবং **price** নামে ২ টা ক্লাস ভেরিয়েবল তৈরি হয়েছে

যাদের মান হিসেবে কন্সট্রাক্টর এর প্যারামিটার এর মান এসাইন হয়েছে। এবং এইযে আমরা **`self`** ইউস করছি এই সেলফ টা তার সেই অবজেক্টটা নির্দেশ করে যেই অবজেক্ট থেকে এটাকে কল করা হয়েছে যেমন **`samsung`** অবজেক্ট এর **`self`** মানে হচ্ছে **`samsung`** অবজেক্টটা নিজেই এবং **`iphone`** অবজেক্ট এর **`self`** মানে হচ্ছে **`iphone`** অবজেক্টটা নিজেই

> ৫.৪ঃ ক্লাস এট্রিবিউট ভার্সেস ইন্সট্যান্স এট্রিবিউট

class attribute v/s inheritance attribute

পাইথনে অবজেক্ট ওরিয়েন্টেড প্রোগ্রামিং নিয়ে কাজ করার জন্য ক্লাস এট্রিবিউট এবং ইন্সট্যান্স এট্রিবিউট এর ব্যাপারে জানাটা ভীষণ জরুরি।

ক্লাস এট্রিবিউটঃ (class attribute)

ক্লাস এট্রিবিউট বোঝার জন্য নিচের উদাহরনটা লক্ষ করি—

```
class Shop:
    cart = []

    def __init__(self, buyer):
        self.buyer = buyer

    def add_to_cart(self, product):
        self.cart.append(product)

mamun = Shop("Mamun")
mamun.add_to_cart("Shoes")
mamun.add_to_cart("Shirt")
print("After mamun added:", mamun.cart)

mahmud = Shop("Mahmud")
mahmud.add_to_cart('Cap')
mahmud.add_to_cart("Watch")
print("After mahmud added:", mahmud.cart)
```

এই কোডটাকে যদি রান করি তাহলে এরকম আউটপুট পাবো

```
After mamun added: ['Shoes', 'Shirt']
After mahmud added: ['Shoes', 'Shirt', 'Cap', 'Watch']
```

কিন্তু আউটপুট এবং কোড ২ টা যদি আমরা খেয়াল করি তাহলে দেখব যে মাহমুদ **add_to_cart()** করার পরে যদিও আমরা **mahmud.cart** এইভাবে ভ্যালুগুলোকে প্রিন্ট করছি তারপরও সেখানে আগের অবজেক্ট এর এড করা প্রোডাক্ট ও রয়ে গেছে এই জিনিসটাই হচ্ছে ক্লাস এট্রিবিউট অর্থাৎ এই এট্রিবিউট টা এই ক্লাস এর প্রোপার্টি এই ক্লাস এর যেকোনো এট্রিবিউট দিয়ে যদি এই এট্রিবিউট এ কোনো চেঞ্জ হয় সেটা ডিরেক্ট এই এট্রিবিউট এই চেঞ্জ করবে

ইন্সট্যান্স এট্রিবিউটঃ

এখন দেখি ইন্সট্যান্স এট্রিবিউট কিভাবে কাজ করে

```
class Shop:

    def __init__(self, buyer):
        self.buyer = buyer
        self.cart = []

    def add_to_cart(self, product):
        self.cart.append(product)

mamun = Shop("Mamun")
mamun.add_to_cart("Shoes")
mamun.add_to_cart("Shirt")
print("After mamun added:", mamun.cart)

mahmud = Shop("Mahmud")
mahmud.add_to_cart('Cap')
mahmud.add_to_cart("Watch")
print("After mahmud added:", mahmud.cart)
```

এই কোডটাকে যদি রান করি তাহলে এরকম আউটপুট পাবো

```
After mamun added: ['Shoes', 'Shirt']
After mahmud added: ['Cap', 'Watch']
```

এইবার দেখুন যখন যেই অবজেক্ট যেটা যেটা **`add_to_cart`** এ এড করছে সেটা শুধু সেই অবজেক্ট এর আন্ডারেই এড হয়ছে

> ৫-৫ঃ ক্লাস ইউস করে ব্যাংক এর উইথড্র, ডিপোজিট এবং ব্যালেন্স এক্সপ্লোর করা

Bank withdraw, deposit, balance using class

এখন পর্যন্ত আমরা যেই কনসেপ্টগুলো শিখছি সেগুলো দিয়ে আমারা একটা ব্যাংক এর উইথড্র, ডিপোজিট এবং ব্যালেন্স এক্সপ্লোর করার ফাংশনালিটি বানাতে পারি এইভাবে

```
class Bank:
    def __init__(self, balance):
        self.balance = balance
        self.min_withdraw = 100
        self.max_withdraw = 100000

    def get_balance(self):
        return self.balance

    def deposit(self, amount):
        if amount > 0:
            self.balance += amount
```

```
def withdraw(self, amount):
        if amount < self.min_withdraw:
            print(f'fokira. you can withdraw below
            {self.min_withdraw}')

        elif amount > self.max_withdraw:
            print (f'bank fokir hoye jabe. you can not with
            more than {self.max_withdraw}')

        else:
            self.balance -= amount
            print(f'Here is your money {amount}')
            print(f'your balance after withdraw:
            {self.get_balance()}')


brac = Bank(15000)
brac.withdraw(25)
brac.withdraw(50000000)
brac.withdraw(1000)

dbbl = Bank(5000)
dbbl.deposit(2000)
dbbl.deposit(2000)
print(dbbl.get_balance())
```

এখানে আমরা ব্যংক নামে একটা ক্লাস ডিক্লেয়ার করে তারমধ্যে কিছু ইন্সট্যান্স এট্রিবিউট এবং প্রতিটা ফাংশনালিটির জন্য আলাদা আলাদা মেথড ডিক্লেয়ার করছি। এখানে মেথড এর মধ্যে আমরা যেভাবে একটা ফাংশন এর মধ্যে কাজগুলো করতাম সেভাবেই কাজগুলো করা হয়েছে।

!! এখন তোমার কাজ হবে সিমিলার টাইপ এর একটা আইডিয়া জেনেরেট করে সেটা ইমপ্লিমেন্ট করার চেষ্টা করা।

> ৫-৬ঃ শপিং চেক-আউট এবং প্রাইস ক্যালকুলেশন

Shopping checkout & price calculation

এই মডিউল এ আমরা আরেকটা OOP এর এক্সাম্পল দেখব। যেমন আমরা একটা শপিং চেক-আউট এবং প্রাইস ক্যালকুলেশন ফাংশনালিটি ফাংশন গুলো এইভাবে লিখতে পারি

```
class Shopping:
    def __init__(self, name):
        self.name = name
        self.cart = []

    def add_to_cart(self, item, price, quantity):
        product ={'item': item, 'price': price, ‘quantity':
        quantity}
        self.cart.append(product)

    # homework
    def remove_item(self, item):
        pass

    def checkout(self, amount):
        total = 0
        for item in self.cart:
            # print(item)
            total += item['price'] * item['quantity']
        print('total price', total)
        if amount < total:
            print(f'Please provide {total -amount} more')
        else:
            extra = amount - total
            print(f'Here is your items and extra money:
            {extra}’)
```

```
swapan = Shopping('Alan Swapon')
swapan.add_to_cart('alu', 50, 6)
swapan.add_to_cart('dim', 12, 24)
swapan.add_to_cart('rice', 50, 5)

print(swapan.cart)
# swapan.checkout(600)
swapan.checkout(1600)
```

এখানে আমরা এড টু কার্ট এবং চেকআউট ফাংশনালিটি গুলো আমরা মেথড এর মাধ্যমে এড করছি এবং চেকয়াউট মেথড এর মধ্যে total বিলটা আমরা কার্ট এর উপর লুপ চালিয়ে সবগুলো প্রোডাক্ট এর প্রাইস এড করে বের করছি এবং সেই অনুযায়ী অন্যান্য অপারেশন করা হয়েছে

!! এখানে তোমার কাজ হচ্ছে remove_item() ফাংশনালিটিটা এড করা যেখানে একটা প্রোডাক্ট এর নাম দিলে সেটাকে কার্ট থেকে রিমুভ করে দিব।

> ৫-৭ঃ মাল্টিপল ক্লাস ব্যবহার করে স্কুল ক্রিয়েট করা

Using Multiple class create school

এই মডিউল এ আমরা মাল্টিপল ক্লাস ইউস করে ছোট একটা স্কুল প্রজেক্ট তৈরি করব
শুরুতেই Stdudent এবং টিচার এর জন্য ২ টা আলাদা ক্লাস বানিয়ে নেয়।

```
class Student:
    def __init__(self, name, current_class, id):
        self.name = name
        self.id = id
        self.current_class = current_class

    def __repr__(self) → str:
        return f'Student with name: {self.name}, class:
        {self.current_class}, id:{self.id}'

class Teacher:
    def __init__(self, name, subject, id) → None:
        self.name = name
        self.subject = subject
        self.id = id

    def __repr__(self) → str:
        return f'Teacher: {self.name}, subject:
        {self.subject}'
```

!! এখানে ২ টা ক্লাস এর মধ্যেই যেই **`__repr__`** মেথড ব্যবহার করা হয়েছে এটা ক্লাস অবজেক্ট এর স্ট্রিং রিপ্রেজেন্টেশন দিবে অর্থাৎ আমরা যদি `Teacher` ক্লাস এর অবজেক্ট এর প্রিন্ট করি তাহলে এটা এই মেথড এর মধ্যে যেই স্ট্রিংটা রিটার্ন করা হচ্ছে সেই অনুযায়ী আউটপুট প্রিন্ট করবে

এবার আমরা School নামে একটা ক্লাস তৈরি করি

```
class School:
    def __init__(self, name) → None:
        self.name = name
        self.teachers = []
        self.students = []

    def add_teacher(self, name, subject):
        id = len(self.teachers) + 101
        teacher = Teacher(name, subject, id)
        self.teachers.append(teacher)

    def enroll(self, name, fee):
        if fee < 6500:
            return 'not enough fee’
        else:
            id = len(self.students) + 1
            student = Student(name, 'C', id)
            self.students.append(student)
            return f'{name} is enrolled with id: {id}, extra
            money {fee - 6500}'

    def __repr__(self) → str:
        print('welcome to', self.name)
        print('————————OUR Teachers————————')
        for teacher in self.teachers:
            print(teacher)
        print('————————OUR STUDENTS————————')
        for student in self.students:
            print(student)
        return 'All Done for now’
```

এখানে শুরুতে আমরা name, teacher এবং students নামে তিনটা ইন্সট্যান্স এট্রিবিউট ডিক্লেয়ার করে নিয়েছি। তারপর মেথড তৈরি করেছি

add_teacher()

এই মেথডের মধ্যে আমরা টিচার এর আইডি হিসিবে টিচার লিস্ট এর লেন্থ এর সাথে ১০১ যোগ করে দিছি আইডিটাকে ইউনিক রাখার জন্য তারপরে এই মেথডের মধ্যে টিচার ক্লাস এর একটা ইন্সট্যান্স তৈরি করা হয়েছে teacher নামে তারপর এই ইন্সট্যান্সটাকে স্কুল ক্লাস এর টিচার লিস্টের মধ্যে এপেন্ড করে দেওয়া হয়েছে।

enroll()

এই মেথডের মধ্যে শুরুতে চেক করা হচ্ছে ফি টা ৬৫০০ এর থেকে কম আছে কিনা যদি থাকে তাহলে সেই অনুযায়ী একটা মেসেজ রিটার্ন করবে

এবং যদি ফি ঠিক থাকে তাহলে টিচার এর মতোই স্টুডেন্ট এর একটা আইডি সেট করে দেওয়া হচ্ছে তারপর স্টুডেন্ট ক্লাস এর একটা ইন্সট্যান্স তৈরি করে সেটাকে Student ক্লাস এর students নামে যেই লিস্ট টা ডিফাইন করা হয়েছে সেখানে এপেন্ড করে দেওয়া হয়েছে।

__repr__()

এটা হচ্ছে আমরা যদি School ক্লাস এর একটা অবজেক্ট তৈরি করি এইভাবে

```
school = School("XYZ Govt School")
print(school)
```

তাহলে এই **school** অবজেক্টটা এই এই **__repr__()** মেথডে যেভাবে ডিফাইন করা হয়েছে সেই অনুযায়ী প্রিন্ট করবে

এখন আমরা সম্পূর্ণ কোডটাকে এইভাবে টেস্ট করে দেখতে পারি

```
phitron = School('Phitron')
phitron.enroll('alia', 5200)
phitron.enroll('rani', 8000)
phitron.enroll('aishwaraiya', 7000)
phitron.enroll('vaijaan', 90000)

phitron.add_teacher('Tom Cruise', 'Algo')
phitron.add_teacher('Decap', 'DS')
phitron.add_teacher('AJ', 'Database')

print(phitron)
```

তাহলে এরকম আউটপুট পাবো

```
welcome to Phitron
————————OUR Teachers————————
Teacher: Tom Cruise, subject: Algo
Teacher: Decap, subject: DS
Teacher: AJ, subject: Database
————————OUR STUDENTS————————
Student with name: rani, class: C, id:1
Student with name: aishwaraiya, class: C, id:2
Student with name: vaijaan, class: C, id:3
All Done for now
```

পাইথন OOP

> ৬-১ : ইন্ট্রোডাকশন টু পাইথন OOP

Introduction Python OOP

এই মডিউলে আমরা OOP এর ব্যাসিকস সম্পর্কে জানার চেষ্টা করব।

পূর্বের মডিউল গুলোতে ক্লাস সম্পর্কে জেনেছি ।

এখানে class কিওয়ার্ড ব্যবহার করে আমরা Phone নামে একটা ক্লাস তৈরি করেছি এবং সেই ক্লাস এর একটি অবজেক্ট হচ্ছে myphone এই অবজেক্ট এর মাধ্যমে আমরা myphone এর প্রোপার্টি যেমন price, color, brand এক্সেস করতে পারি।

> ৬-২ : ইনহেরিটেনস

Inheritance

আমরা আগের মডিউল এ একটা সিম্পল ক্লাস ডিক্লেয়ার করেছিলাম যেখানে কিছু ক্লাস ভেরিয়েবল ডিক্লেয়ার করেছিলাম আমরা একটা ক্লাস এর মধ্যে ভেরিয়েবল ছাড়াও মেথড ও ডিক্লেয়ার করতে পারি একটা মেথড কোনো একটা ভেরিয়েবল এর বিহেবিয়ার কে নির্দেশ করে যেমন আমাদের Phone ক্লাস এর একটা বিহেবিয়ার হতে পারে Call তাহলে এই নামে আমরা একটা মেথড ডিক্লেয়ার করতে পারি এইভাবে

```
class Phone:
    price = 19000
    color = 'blue'
    brand = 'samsung'

    def call(self):
        print('Calling one person')

myphone = Phone()
myphone.call()

#output: Calling one person
```

এখানে **`call()`** নামে যেই ফাংশন এর মতো ডিক্লেয়ার করা হয়েছে ক্লাস এর মধ্যে এটাই মেথড। ফাংশন যখন কোনো একটা ক্লাস এর আন্ডারে ডিফাইন করা হয় তখন সেটাকে মেথড বলে এবং এই মেথডটি প্রতিটি অবজেক্ট এর জন্য স্বতন্ত্রভাবে কাজ করে যেমন **`myphone.call()`** এর মাধ্যমে **`myphone`** নামের অবজেক্ট এর **`call`** মেথডকে কল করা হয়েছে।

!! মনে রাখতে হবে পাইথনে মেথড ডিক্লেয়ার করার ক্ষেত্রে প্রথম প্যারামিটার হিসেবে self দিতে হবে

এবং আমরা চাইলে একটা ক্লাস এর মধ্যে মাল্টিপল মেথডও ক্রিয়েট করতে পারি এইভাবে

```
class Phone:
    price = 19000
    color = 'blue'
    brand = 'samsung'

    def call(self):
        print('Calling one person')

    def send_message(self, message):
        return f"Sending message: {message}"

myphone = Phone()
myphone.call() #output: Calling one person
print(myphone.send_message("Hello World"))

#output:Sending message: Hello World
```

একটা ফাংশন এর মতো করে আমরা মেথড এও প্যারামিটার নিতে পারি এবং মেথড থেকে কোনোকিছু রিটার্ন ও করতে পারি

Inheritance Concept:

```
# base class, parent class
class BaseClass:
    pass

# derived class or child class
class DerivedClass(BaseClass):
    pass


"""
1. simple inheritance: parent class ⟶ child class (Gadget ⟶ Phone) (Gadget ⟶ Laptop)

2. Multi-level inheritance: Granda ⟶ Parent ⟶ child (Vehicle ⟶ Bus ⟶ ACBus) (Vehicle ⟶ Truck ⟶ PickupTruck)

3. Multiple inheritance: Student (Family, School, Sports)

4. Hybrid: Granda ⟶ Father, Uncle, Aunty ⟶ Child (Father, Uncle)
"""
```

> ৬-৩ : মাল্টি-লেভেল ইনহেরিটেনস

Multi Level Inheritance

মাল্টি-লেভেল ইনহেরিটেন্স (Multi-level Inheritance) কী?

মাল্টি-লেভেল ইনহেরিটেন্স হল একটি ইনহেরিটেন্সের প্রকার যেখানে একটি ক্লাস অন্য একটি ক্লাস থেকে বৈশিষ্ট্য ও মেথডগুলো উত্তরাধিকার সূত্রে পায়, এবং সেই ক্লাসটি আবার অন্য একটি ক্লাস থেকে বৈশিষ্ট্য ও মেথডগুলো উত্তরাধিকার সূত্রে পায়।

এটি একটি চেইন-এর মত কাজ করে যেখানে প্রতিটি ক্লাস তার পূর্ববর্তী ক্লাসের বৈশিষ্ট্য ও মেথডগুলো গ্রহণ করে।

উদাহরণ:

আমাদের উদাহরণে তিনটি ক্লাস রয়েছে: A, B, এবং C।

A হল মূল বা বেস ক্লাস।
B ক্লাসটি A ক্লাস থেকে ইনহেরিট করে।
C ক্লাসটি B ক্লাস থেকে ইনহেরিট করে।
এইভাবে, C ক্লাসটি A এবং B উভয়ের বৈশিষ্ট্য ও মেথডগুলো পাবে।

```
class A:
    def __init__(self, name):
        self.name = name

    def display_name(self):
        print(f'Name: {self.name}')

class B(A):
    def __init__(self, name, age):
        super().__init__(name)
        self.age = age

    def display_age(self):
        print(f'Age: {self.age}')

class C(B):
    def __init__(self, name, age, grade):
        super().__init__(name, age)
        self.grade = grade

    def display_grade(self):
        print(f'Grade: {self.grade}')
```

```
# C ক্লাসের একটি অবজেক্ট তৈরি করা হল
student = C('John', 20, 'A')

# সমস্ত ক্লাসের মেথডগুলো ব্যবহার করা হচ্ছে
student.display_name()  # Output: Name: John
student.display_age()   # Output: Age: 20
student.display_grade() # Output: Grade: A
```

ক্লাস A:
- **Constructor Method (__init__):** name সেট করে।
- **Method (display_name):** name প্রিন্ট করে।

ক্লাস B (উত্তরাধিকারসূত্রে A থেকে):
- **Constructor Method (__init__):** name এবং age সেট করে।
- **super().__init__(name):** বেস ক্লাস A এর constructor কে কল করে।
- **Method (display_age):** age প্রিন্ট করে।

ক্লাস C (উত্তরাধিকারসূত্রে B থেকে):
- **Constructor Method (__init__):** name, age, এবং grade সেট করে।
- **super().__init__(name, age):** পেরেন্ট ক্লাস B এর constructor কে কল করে।
- **Method (display_grade):** grade প্রিন্ট করে।

মাল্টি-লেভেল ইনহেরিটেন্সের সুবিধা:

1. **কোড পুনর্ব্যবহারযোগ্যতা বৃদ্ধি:** একই কোড বিভিন্ন ক্লাসে পুনরায় ব্যবহার করা যায়।
2. **সংগঠিত ও মডুলার কোড:** কোডকে ছোট ছোট অংশে বিভক্ত করে ভালোভাবে পরিচালনা করা যায়।
3. **সহজ রক্ষণাবেক্ষণ:** কোডের পরিবর্তন ও আপডেট সহজ হয়।

এইভাবে, মাল্টি-লেভেল ইনহেরিটেন্স প্রোগ্রামিংয়ে কার্যকরী এবং সংগঠিত কোড লেখার জন্য একটি গুরুত্বপূর্ণ কৌশল

Example of multi level inheritance:

`Vehicle` ক্লাস: এটি মূল বা `base` ক্লাস। এখানে দুটি বৈশিষ্ট্য (`attribute`) এবং একটি মেথড (`method`) রয়েছে:

- `__init__(self, name, price)`: এটি `constructor` মেথড যা `name` এবং `price attribute` সেট করে।
- `__repr__(self)`: এটি একটি মেথড যা ক্লাসের অবজেক্টের একটি `readable` স্ট্রিং রিটার্ন করে।
- `move(self)`: এটি একটি ফাঁকা মেথড যা পরবর্তীতে `subclass`-এ ওভাররাইড করা যেতে পারে।

Step 1: Base Class - Vehicle

```
class Vehicle:
    def __init__(self, name, price) → None:
        self.name = name
        self.price = price

    def __repr__(self) → str:
        return f'{self.name} {self.price}'

    def move(self):
        pass
```

Bus ক্লাস: এটি Vehicle ক্লাস থেকে উত্তরাধিকার সূত্রে প্রাপ্ত। এখানে নতুন একটি attribute seat যুক্ত করা হয়েছে এবং Vehicle-এর constructor কে super() ফাংশনের মাধ্যমে কল করা হয়েছে।

Step 2: Derived Class - Bus

```
class Bus(Vehicle):
    def __init__(self, name, price, seat) → None:
        self.seat = seat
        super().__init__(name, price)

    def __repr__(self) → str:
        return super().__repr__()
```

Truck ক্লাস: এটি Vehicle ক্লাস থেকে উত্তরাধিকার সূত্রে প্রাপ্ত। এখানে নতুন একটি attribute weight যুক্ত করা হয়েছে এবং Vehicle-এর constructor কে super() ফাংশনের মাধ্যমে কল করা হয়েছে।

Step 3: Derived Class - Truck

```
class Truck(Vehicle):
    def __init__(self, name, price, weight) → None:
        self.weight = weight
        super().__init__(name, price)
```

PickUpTruck ক্লাস: এটি Truck ক্লাস থেকে উত্তরাধিকার সূত্রে প্রাপ্ত। এটি Truck-এর constructor কে super() ফাংশনের মাধ্যমে কল করে।

Step 4: Derived Class - PickUpTruck

```
class PickUpTruck(Truck):
    def __init__(self, name, price, weight) → None:
        super().__init__(name, price, weight)
```

ACBus ক্লাস: এটি Bus ক্লাস থেকে উত্তরাধিকার সূত্রে প্রাপ্ত। এখানে নতুন একটি attribute temperature যুক্ত করা হয়েছে এবং Bus-এর constructor কে super() ফাংশনের মাধ্যমে কল করা হয়েছে।

এছাড়াও, __repr__(self) মেথডটি ওভাররাইড করা হয়েছে যা প্রথমে seat attribute প্রিন্ট করে তারপর Vehicle-এর __repr__ মেথড কল করে।

Step 5: Derived Class - ACBus

```
class ACBus(Bus):
    def __init__(self, name, price, seat, temperature) →
None:
        self.temperature = temperature
        super().__init__(name, price, seat)
    def __repr__(self) → str:
        print(f'{self.seat}')
        return super().__repr__()
```

ACBus অবজেক্ট তৈরি এবং প্রিন্ট করা:

- green_line নামের একটি ACBus অবজেক্ট তৈরি করা হয়েছে যার name 'green', price 5000000, seat 22, এবং temperature 16।
- print(green_line) কল করলে, __repr__() মেথড কল হবে যা seat প্রিন্ট করবে এবং তারপর name এবং price প্রিন্ট করবে।

Step 6: Object Creation and Method Call

```
green_line = ACBus('green', 5000000, 22, 16)
print(green_line)


# output:
22
green 5000000
```

Multiple Inheritance

মাল্টিপল ইনহেরিটেন্স (Multiple Inheritance) কি?

মাল্টিপল ইনহেরিটেন্স হল প্রোগ্রামিং-এর একটি কৌশল যেখানে একটি ক্লাস একাধিক ক্লাস থেকে বৈশিষ্ট্য ও মেথডগুলো উত্তরাধিকার সূত্রে পায়। এতে করে একটি সাবক্লাস একাধিক পেরেন্ট ক্লাসের বৈশিষ্ট্য ও মেথডগুলো ব্যবহার করতে পারে।

```
class Family:
    def __init__(self, address) → None:
        self.address = address

class School:
    def __init__(self, id, level) → None:
        self.id = id
        self.level = level

class Sports:
    def __init__(self, game) → None:
        self.game = game

class Student(Family, School, Sports):
    def __init__(self, address, id, level, game) → None:
        School.__init__(self, id, level)
        Sports.__init__(self, game)
        Family.__init__(self, address)
```

```
student = Student('123 Main St', 101, 'High', 'Football')

print(student.address)  # Output: 123 Main St
print(student.id)       # Output: 101
print(student.level)    # Output: High
print(student.game)     # Output: Football
```

Encapsulation & Access Modifiers (private, Public, Protected)

এনক্যাপসুলেশন (Encapsulation):

এনক্যাপসুলেশন হল প্রোগ্রামিংয়ের একটি ধারণা যেখানে ডাটা এবং মেথডগুলোকে একত্রিত করে একটি ইউনিট বা ক্লাসে আবদ্ধ করা হয়। এটি ডাটা হাইডিং নিশ্চিত করে, অর্থাৎ ডাটাকে ক্লাসের বাহির থেকে সরাসরি অ্যাক্সেস করা থেকে রক্ষা করে।

অ্যাক্সেস মডিফায়ারস (Access Modifiers):

Python-এ তিন ধরণের অ্যাক্সেস মডিফায়ারস রয়েছে:

Public:

- পাবলিক অ্যাট্রিবিউট এবং মেথডগুলো ক্লাসের বাইরে থেকে সরাসরি অ্যাক্সেস করা যায়।
- উদাহরণ: self.holder_name

```
class MyClass:
    def __init__(self, name):
        self.name = name  # Public attribute

    def display(self):
        print(self.name)  # Public method

obj = MyClass('Alice')
print(obj.name)  # Accessing public attribute
obj.display()    # Calling public method
```

Protected:

• প্রটেক্টেড অ্যাট্রিবিউট এবং মেথডগুলো শুধুমাত্র ক্লাস এবং এর সাবক্লাসের মধ্যে অ্যাক্সেসযোগ্য।
• `_` দিয়ে শুরু হয়।
• উদাহরণ: self._branch

```
class MyClass:
    def __init__(self, name):
        self._name = name  # Protected attribute

    def _display(self):
        print(self._name)  # Protected method

class SubClass(MyClass):
    def show(self):
        print(self._name)  # Accessing protected attribute
        self._display()    # Calling protected method

obj = SubClass('Bob')
obj.show()
```

Private:

• প্রাইভেট অ্যাট্রিবিউট এবং মেথডগুলো শুধুমাত্র ক্লাসের ভিতরে অ্যাক্সেসযোগ্য।
• `__` দিয়ে শুরু হয়।
• উদাহরণ: self.__balance

```
class MyClass:
    def __init__(self, name):
        self.__name = name  # Private attribute

    def __display(self):
        print(self.__name)  # Private method

    def show(self):
        print(self.__name)  # Accessing private attribute
within the class
        self.__display()    # Calling private method within
the class

obj = MyClass('Charlie')
obj.show()

# print(obj.__name)  # Error: Can't access private attribute
# obj.__display()    # Error: Can't call private method
```

সংক্ষেপে ব্যাখ্যা:

Public: ক্লাসের বাহির থেকে সরাসরি অ্যাক্সেসযোগ্য।
Protected: শুধুমাত্র ক্লাস এবং এর সাবক্লাসের মধ্যে অ্যাক্সেসযোগ্য।
Private: শুধুমাত্র ক্লাসের ভিতরে অ্যাক্সেসযোগ্য।

এনক্যাপসুলেশন এবং অ্যাক্সেস মডিফায়ারস ব্যবহার করে আমরা ডাটার সুরক্ষা নিশ্চিত করতে পারি এবং অবাঞ্ছিত অ্যাক্সেস প্রতিরোধ করতে পারি। এটি প্রোগ্রামিংয়ে একটি গুরুত্বপূর্ণ ধারণা যা কোডের গঠন এবং রক্ষণাবেক্ষণ সহজ করে।

Example

এখানে একটি `Bank` ক্লাস তৈরি করা হয়েছে যা এনক্যাপসুলেশন এবং অ্যাক্সেস মডিফায়ারস ব্যবহার করে। আমরা দেখব কীভাবে পাবলিক, প্রটেক্টেড, এবং প্রাইভেট অ্যাট্রিবিউট ব্যবহার করা হয়েছে।

```
class Bank:
    def __init__(self, holder_name, initial_deposit) → None:
        self.holder_name = holder_name   # Public attribute
        self._branch = 'banani 11'       # Protected attribute
        self.__balance = initial_deposit # Private attribute

    def deposit(self, amount):
        self.__balance += amount

    def get_balance(self):
        return self.__balance

    def withdraw(self, amount):
        if amount < self.__balance:
            self.__balance -= amount
            return amount
        else:
            return f'Forkia taka nai'
```

```
# একটি Bank ক্লাসের অবজেক্ট তৈরি করা হয়েছে
rafsun = Bank('Choooto bro', 10000)

# Public attribute অ্যাক্সেস করা হচ্ছে
print(rafsun.holder_name)  # Output: Choooto bro

# Public attribute পরিবর্তন করা হচ্ছে
rafsun.holder_name = 'boro vai'
```

```
# Deposit method ব্যবহার করে balance বাড়ানো হচ্ছে
rafsun.deposit(40000)

# Balance চেক করা হচ্ছে
print(rafsun.get_balance())  # Output: 50000

# Public attribute আবার চেক করা হচ্ছে
print(rafsun.holder_name)    # Output: boro vai
```

```
# Protected attribute (uncomment করে দেখা যেতে পারে)
# print(rafsun._branch)

# Private attribute অ্যাক্সেস করা হচ্ছে (indirect method ব্যবহার করে)
# print(dir(rafsun)) # Uncomment to see all attributes

print(rafsun._Bank__balance)  # Output: 50000
```

> ৬-৬ : অ্যাবস্ট্র্যাক্ট ক্লাস ও অ্যাবস্ট্র্যাক্ট মেথড

Abstract Classes & Abstract Method

ইন্সট্যান্সিয়েট (Instantiate)

ইন্সট্যান্সিয়েট (Instantiate) হলো একটি ক্লাস থেকে একটি অবজেক্ট তৈরি করার প্রক্রিয়া। পাইথনে, একটি ক্লাস হলো একটি টেমপ্লেট যা একাধিক অবজেক্ট তৈরি করতে ব্যবহৃত হয়। যখন আপনি একটি ক্লাস থেকে একটি অবজেক্ট তৈরি করেন, তখন সেই প্রক্রিয়াটিকে ইন্সট্যান্সিয়েশন বলা হয়।

```
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def bark(self):
        return f"{self.name} is barking!"

# Dog ক্লাস থেকে একটি অবজেক্ট তৈরি করা হচ্ছে
my_dog = Dog("Buddy", 3)

print(my_dog.name)  # Output: Buddy
print(my_dog.age)   # Output: 3
print(my_dog.bark())  # Output: Buddy is barking!
```

এখানে, Dog হলো একটি ক্লাস যা name এবং age প্রোপার্টি এবং একটি bark মেথড দিয়ে ডিফাইন করা হয়েছে। my_dog = Dog("Buddy", 3) এই লাইনটি Dog ক্লাস থেকে একটি নতুন অবজেক্ট তৈরি করছে, এবং এই প্রক্রিয়াটিকে ইন্সট্যান্সিয়েশন বলা হয়।

সংক্ষেপে বলতে গেলে, ইন্সট্যান্সিয়েট করা মানে হলো:

1. একটি ক্লাস ডিফাইন করা।
2. সেই ক্লাস থেকে একটি নির্দিষ্ট অবজেক্ট তৈরি করা।

অ্যাবস্ট্রাক্ট ক্লাস

অ্যাবস্ট্রাক্ট ক্লাস হল এমন একটি ক্লাস যা কখনও সরাসরি ইন্সট্যান্সিয়েট করা যায় না। এটি মূলত অন্যান্য ক্লাসের জন্য একটি বেস বা প্যারেন্ট ক্লাস হিসাবে কাজ করে।

অ্যাবস্ট্রাক্ট মেথড

অ্যাবস্ট্রাক্ট মেথড হল এমন একটি মেথড যা শুধুমাত্র ডিক্লেয়ার করা হয় কিন্তু এর কোনো ইমপ্লিমেন্টেশন থাকে না। সাবক্লাসগুলোতে এই মেথডগুলি ওভাররাইড করতে বাধ্য করতে ব্যবহৃত হয়।

Python-এ অ্যাবস্ট্রাক্ট ক্লাস এবং মেথড:

Python-এ অ্যাবস্ট্রাক্ট ক্লাস এবং মেথডগুলি তৈরি করার জন্য abc (Abstract Base Classes) মডিউল ব্যবহার করা হয়।

Step 1: `abc` মডিউল ইমপোর্ট করা

```
from abc import ABC, abstractmethod
```

Animal ক্লাসটি ABC থেকে ইনহেরিট করে, যা এটিকে একটি অ্যাবস্ট্রাক্ট ক্লাস করে।
sound এবং move মেথডগুলি **@abstractmethod** ডেকোরেটরের মাধ্যমে অ্যাবস্ট্রাক্ট করা হয়েছে।

Step 2: অ্যাবস্ট্রাক্ট ক্লাস তৈরি করা

```
class Animal(ABC):
    @abstractmethod
    def sound(self):
        pass

    @abstractmethod
    def move(self):
        pass
```

Dog এবং Bird ক্লাসগুলি Animal ক্লাস থেকে ইনহেরিট করে।
sound এবং move মেথডগুলি প্রতিটি সাবক্লাসে ওভাররাইড করা হয়েছে।

Step 3: সাবক্লাস তৈরি করা এবং অ্যাবস্ট্রাক্ট মেথড ওভাররাইড করা

```
class Dog(Animal):
    def sound(self):
        return "Bark"

    def move(self):
        return "Run"

class Bird(Animal):
    def sound(self):
        return "Chirp"

    def move(self):
        return "Fly"
```

Step 4: অবজেক্ট তৈরি করা এবং মেথড কল করা

```
dog = Dog()
bird = Bird()

print(dog.sound())  # Output: Bark
print(dog.move())   # Output: Run

print(bird.sound()) # Output: Chirp
print(bird.move())  # Output: Fly
```

সংক্ষেপে:

1. **Abstract Class:** ABC মডিউল থেকে ইনহেরিট করে তৈরি করা হয়।
2. **Abstract Method:** @abstractmethod ডেকোরেটর দিয়ে ডিক্লেয়ার করা হয় এবং এর কোনো ইমপ্লিমেন্টেশন থাকে না।
3. **Subclass:** অ্যাবস্ট্রাক্ট ক্লাস থেকে ইনহেরিট করে এবং অ্যাবস্ট্রাক্ট মেথডগুলি ওভাররাইড করে।

অ্যাবস্ট্রাক্ট ক্লাস এবং মেথডগুলি ব্যবহারের মূল উদ্দেশ্য হলো **একটি স্ট্যান্ডার্ড ইন্টারফেস তৈরি করা** যা সব সাবক্লাসে ইমপ্লিমেন্ট করা বাধ্যতামূলক।

```
from abc import ABC, abstractmethod
# abstract base class

class Animal(ABC):
    @abstractmethod #enforce all derived class to have a eat
                    method

    def eat(self):
        print('I need food')

    @abstractmethod
    def move(self):
        pass

class Monkey(Animal):
    def __init__(self, name) → None:
        self.category = 'Monkey'
        self.name = name
        super().__init__()

    def eat(self):
        print('Hey na nana, I am eating banana')
    def move(self):
        print('Hanging on the branches')

layka = Monkey('lucky')
layka.eat()
```

Abstract Class & Interface

অ্যাবস্ট্রাক্ট ক্লাস (Abstract Class):

অ্যাবস্ট্রাক্ট ক্লাস হল এমন একটি ক্লাস যা কখনো সরাসরি ইন্সট্যান্সিয়েট করা যায় না। এটি অন্যান্য ক্লাসের জন্য একটি বেস বা প্যারেন্ট ক্লাস হিসেবে কাজ করে। এতে অ্যাবস্ট্রাক্ট মেথড থাকে যা শুধুমাত্র ডিক্লেয়ার করা হয়, কিন্তু ইমপ্লিমেন্ট করা হয় না।

মূল বৈশিষ্ট্য:

- ABC মডিউল থেকে ইনহেরিট করে।
- `@abstractmethod` ডেকোরেটর দিয়ে অ্যাবস্ট্রাক্ট মেথড তৈরি করা হয়।
- সাবক্লাসগুলি এই মেথডগুলি ওভাররাইড করতে বাধ্য।

```
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def eat(self):
        pass

    @abstractmethod
    def move(self):
        pass

class Dog(Animal):
    def eat(self):
        print("Dog is eating")

    def move(self):
        print("Dog is running")

dog = Dog()
dog.eat()  # Output: Dog is eating
dog.move() # Output: Dog is running
```

ইন্টারফেস (Interface):

Python-এ সরাসরি "ইন্টারফেস" কনসেপ্ট নেই, তবে অ্যাবস্ট্রাক্ট ক্লাস এবং মেথড ব্যবহার করে ইন্টারফেসের মতো ফাংশনালিটি তৈরি করা যায়। ইন্টারফেস হল এমন একটি কাঠামো যা শুধুমাত্র মেথডের ঘোষণা দেয়, কিন্তু ইমপ্লিমেন্টেশন দেয় না। ইন্টারফেস ব্যবহার করে আমরা নির্দিষ্ট মেথডগুলো সব ক্লাসে ইমপ্লিমেন্ট করতে বাধ্য করতে পারি।

মূল বৈশিষ্ট্য:

- সব মেথড অ্যাবস্ট্রাক্ট থাকে।
- কোনো মেথডের ইমপ্লিমেন্টেশন থাকে না।
- অন্যান্য ক্লাসে নির্দিষ্ট ফাংশনালিটি নিশ্চিত করতে ব্যবহৃত হয়।

```
from abc import ABC, abstractmethod

class AnimalInterface(ABC):
    @abstractmethod
    def eat(self):
        pass

    @abstractmethod
    def move(self):
        pass

class Cat(AnimalInterface):
    def eat(self):
        print("Cat is eating")

    def move(self):
        print("Cat is running")

cat = Cat()
cat.eat()  # Output: Cat is eating
cat.move() # Output: Cat is running
```

সংক্ষেপে:

- **অ্যাবস্ট্রাক্ট ক্লাস:** এটি একটি বেস ক্লাস যা অন্যান্য ক্লাসে নির্দিষ্ট মেথড ওভাররাইড করতে বাধ্য করে। এতে কিছু ইমপ্লিমেন্টেশন থাকতে পারে।
- **ইন্টারফেস:** এটি শুধুমাত্র মেথডের ঘোষণা দেয় এবং কোনো ইমপ্লিমেন্টেশন থাকে না। এটি অন্যান্য ক্লাসে নির্দিষ্ট ফাংশনালিটি নিশ্চিত করে।

অ্যাবস্ট্রাক্ট ক্লাস এবং ইন্টারফেস উভয়ই একটি স্ট্যান্ডার্ড ইন্টারফেস তৈরি করতে ব্যবহৃত হয় যা সব সাবক্লাসে ইমপ্লিমেন্ট করতে বাধ্য।

> ৬-৮ : পলিমরফিজম

Polymorphism

পলিমরফিজম (Polymorphism) কি?

পলিমরফিজম শব্দটি এসেছে গ্রিক শব্দ থেকে, যার অর্থ "বহু রূপ"। প্রোগ্রামিংয়ে, পলিমরফিজম বলতে বোঝায় একই ইন্টারফেস বা মেথডের বিভিন্ন বাস্তবায়ন। এটি একটি অবজেক্ট ওরিয়েন্টেড প্রোগ্রামিং (OOP) এর মৌলিক ধারণা।

পলিমরফিজমের উদাহরণ Python এ:

Animal ক্লাসে **`__init__`** মেথডে name অ্যাট্রিবিউট সেট করা হয়। **`make_sound`** মেথডে একটি সাধারণ বার্তা প্রিন্ট করা হয়।

`Step 1:` বেস ক্লাস তৈরি করা

```
class Animal:
    def __init__(self, name) → None:
        self.name = name

    def make_sound(self):
        print('animal making some sound')
```

এই ক্লাসগুলো Animal ক্লাস থেকে ইনহেরিট করে। প্রতিটি ক্লাসে **__init__** মেথডে name অ্যাট্রিবিউট সেট করা হয় এবং **super().__init__(name)** এর মাধ্যমে বেস ক্লাসের কন্সট্রাক্টর কল করা হয়। প্রতিটি ক্লাসে **make_sound** মেথড ওভাররাইড করা হয়েছে এবং সংশ্লিষ্ট প্রাণীর শব্দ প্রিন্ট করা হয়েছে।

Step 2: ডেরাইভড ক্লাস তৈরি করা এবং মেথড ওভাররাইড করা

```
class Cat(Animal):
    def __init__(self, name) → None:
        super().__init__(name)

    def make_sound(self):
        print('meow meow')

class Dog(Animal):
    def __init__(self, name) → None:
        super().__init__(name)

    def make_sound(self):
        print('gheu gheu')

class Goat(Animal):
    def __init__(self, name) → None:
        super().__init__(name)

    def make_sound(self):
        print('beh beh beh')
```

Cat, Dog, এবং Goat ক্লাসের অবজেক্ট তৈরি করা হয়েছে। প্রতিটি অবজেক্টের **make_sound** মেথড কল করা হয়েছে যা সংশ্লিষ্ট শব্দ প্রিন্ট করেছে।

Step 3: অবজেক্ট তৈরি করা এবং মেথড কল করা

```
don = Cat('Real Don')
don.make_sound()  # Output: meow meow

shepard = Dog('Local Shephard')
shepard.make_sound()  # Output: gheu gheu

mess = Goat('L M')
mess.make_sound()  # Output: beh beh beh

less = Goat('gora gori')
```

একটি animals নামের লিস্ট তৈরি করা হয়েছে যেখানে don, shepard, mess, এবং less অবজেক্ট রাখা হয়েছে। একটি for লুপ ব্যবহার করে প্রতিটি animal এর **make_sound** মেথড কল করা হয়েছে। প্রতিটি animal এর নিজস্ব **make_sound** মেথড ওভাররাইড করা হয়েছে, তাই প্রত্যেকটি নিজস্ব শব্দ প্রিন্ট করেছে।

Step 4: পলিমরফিজম ব্যবহার করে একটি লুপের মাধ্যমে মেথড কল করা

```
animals = [don, shepard, mess, less]
for animal in animals:
    animal.make_sound()
```

```
# Output:

meow meow
gheu gheu
beh beh beh
beh beh beh
```

Shape Calculation:

```
from math import pi
class Shape:
    def __init__(self, name) → None:
        self.name = name

class Rectangle(Shape):
    def __init__(self, name, length, width) → None:
        self.length = length
        self.width = width
        super().__init__(name)

    def area(self):
        return self.length * self.width

class Circle(Shape):
    def __init__(self, name, radius) → None:
        self.radius = radius
        super().__init__(name)

    def area(self):
        return pi * self.radius*self.radius
```

```
# Create instances
rectangle = Rectangle("Rectangle", 10, 5)
circle = Circle("Circle", 7)

# Calculate areas
print(f"Area of {rectangle.name}: {rectangle.area()}")
print(f"Area of {circle.name}: {circle.area()}")
```

```
# Output

Area of Rectangle: 50
Area of Circle: 153.93804002589985
```

সংক্ষেপে পলিমরফিজম:

- পলিমরফিজম হল একই ইন্টারফেস বা মেথডের বিভিন্ন বাস্তবায়ন।
- এটি কোডের পুনঃব্যবহারযোগ্যতা এবং নমনীয়তা বাড়ায়।
- একাধিক ক্লাস একই মেথড ব্যবহার করতে পারে কিন্তু আলাদা আলাদা ভাবে বাস্তবায়ন করতে পারে।

মূল পয়েন্টসমূহ:

একই ইন্টারফেস বা মেথডের বিভিন্ন বাস্তবায়ন:

- পলিমরফিজমের মাধ্যমে, আমরা একই ইন্টারফেস বা মেথড বিভিন্ন ক্লাসে বিভিন্নভাবে বাস্তবায়ন করতে পারি।
- উদাহরণস্বরূপ, `make_sound` মেথডটি `Dog` এবং `Cat` ক্লাসে ভিন্নভাবে কাজ করে।

কোডের পুনঃব্যবহারযোগ্যতা এবং নমনীয়তা বৃদ্ধি:

- পলিমরফিজম কোডের পুনঃব্যবহারযোগ্যতা এবং নমনীয়তা বাড়ায় কারণ একই মেথড বিভিন্ন ক্লাসে ব্যবহার করা যায়।
- এটি কোডের ডুপ্লিকেশন কমায় এবং নতুন ফিচার যোগ করা সহজ করে।

সহজে মেইনটেইনেবল এবং এক্সপেন্ডেবল কোড:

- পলিমরফিজম ব্যবহার করে আমরা সহজে মেইনটেইনেবল এবং এক্সপেন্ডেবল কোড লিখতে পারি।
- এটি কোডকে আরও মডুলার করে তোলে, যা পরিবর্তন এবং এক্সপেনশন সহজ করে।

বিভিন্ন কনটেক্সটে কাজ করতে সক্ষম:

- পলিমরফিজম কোডকে বিভিন্ন কনটেক্সটে কাজ করার ক্ষমতা প্রদান করে।
- উদাহরণস্বরূপ, একই `make_sound` মেথড বিভিন্ন প্রাণীর জন্য বিভিন্ন শব্দ তৈরি করতে পারে।

পলিমরফিজম ব্যবহার করে আমরা সহজে মেইনটেইনেবল এবং এক্সপেন্ডেবল কোড লিখতে পারি যা বিভিন্ন কনটেক্সটে কাজ করতে পারে।

Overloading & Overriding

Python-এ Object-Oriented Programming (OOP) এর দুটি গুরুত্বপূর্ণ ধারণা হল অপারেটর ওভারলোডিং এবং মেথড ওভাররাইডিং।

অপারেটর ওভারলোডিং (Operator Overloading)

অপারেটর ওভারলোডিং হল একটি প্রক্রিয়া যেখানে Python-এর বিল্ট-ইন অপারেটরগুলিকে (যেমন: +, -, *, /) কাস্টম ক্লাসে ব্যবহার করা যায়। এটি আমাদের কাস্টম অবজেক্টগুলির মধ্যে অপারেটর ব্যবহার করে অর্থপূর্ণ ক্রিয়াকলাপ নির্ধারণ করতে দেয়।

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

    def __repr__(self):
        return f"Vector({self.x}, {self.y})"

# ভেক্টর অবজেক্ট তৈরি
v1 = Vector(2, 3)
v2 = Vector(4, 5)

# ভেক্টর যোগ করা
v3 = v1 + v2

print(v3)  # Output: Vector(6, 8)
```

`__init__` মেথড:
এটি একটি কন্সট্রাক্টর যা x এবং y অ্যাট্রিবিউট সেট করে।

`__add__` মেথড:

এটি "+" অপারেটর ওভারলোড করে, যা দুটি ভেক্টর যোগ করার জন্য ব্যবহার করা হয়।

`__repr__` মেথড:
এটি অবজেক্টের একটি স্ট্রিং রিটার্ন করে, যা প্রিন্ট করার সময় দেখা যায়।

সহজ ভাষায়:
অপারেটর ওভারলোডিং ব্যবহার করে আমরা "+" অপারেটরকে দুটি ভেক্টর যোগ করার জন্য ব্যবহার করেছি। এখানে, `v1 + v2` আসলে `v1.__add__(v2)` কে কল করে এবং নতুন ভেক্টর রিটার্ন করে।

মেথড ওভাররাইডিং (Method Overriding)

মেথড ওভাররাইডিং হল একটি প্রক্রিয়া যেখানে একটি সাবক্লাস তার পেরেন্ট ক্লাসের মেথড পুনরায় সংজ্ঞায়িত করে। এর ফলে সাবক্লাসে মেথডটি ভিন্নভাবে কাজ করতে পারে।

```
class Animal:
    def speak(self):
        print("Animal makes a sound")

class Dog(Animal):
    def speak(self):
        print("Dog barks")

class Cat(Animal):
    def speak(self):
        print("Cat meows")

# অবজেক্ট তৈরি
dog = Dog()
cat = Cat()

# মেথড কল
dog.speak()  # Output: Dog barks
cat.speak()  # Output: Cat meows
```

পেরেন্ট ক্লাস Animal:
speak মেথডটি "Animal makes a sound" প্রিন্ট করে।

সাবক্লাস Dog:
speak মেথডটি ওভাররাইড করা হয়েছে, যা "Dog barks" প্রিন্ট করে।

সাবক্লাস Cat:
speak মেথডটি ওভাররাইড করা হয়েছে, যা "Cat meows" প্রিন্ট করে।

সহজ ভাষায়:
মেথড ওভাররাইডিং ব্যবহার করে আমরা পেরেন্ট ক্লাস Animal এর speak মেথডটি Dog এবং Cat ক্লাসে ভিন্নভাবে বাস্তবায়িত করেছি। `dog.speak()` এবং `cat.speak()` কল করলে ভিন্ন ভিন্ন আউটপুট পাওয়া যায়।

সংক্ষেপে:

1. অপারেটর ওভারলোডিং:
- এটি একটি প্রক্রিয়া যেখানে আমরা Python-এর অপারেটরগুলিকে কাস্টম ক্লাসে ব্যবহার করতে পারি।
- উদাহরণ: "+" অপারেটরকে দুটি ভেক্টর যোগ করতে ব্যবহার করা।

1. মেথড ওভাররাইডিং:
- এটি একটি প্রক্রিয়া যেখানে একটি সাবক্লাস তার পেরেন্ট ক্লাসের মেথড পুনরায় সংজ্ঞায়িত করে।
- উদাহরণ: Dog এবং Cat ক্লাসে speak মেথড ওভাররাইড করা।

> 7.0 : static attribute, static method and class method decorator

Static attribute, Method and Class method decorator

Python-এ স্ট্যাটিক অ্যাট্রিবিউট, স্ট্যাটিক মেথড এবং ক্লাস মেথডের ব্যবহার OOP এর একটি গুরুত্বপূর্ণ দিক।

স্ট্যাটিক অ্যাট্রিবিউট (Static Attribute)

স্ট্যাটিক অ্যাট্রিবিউট হল এমন একটি অ্যাট্রিবিউট যা ক্লাসের সব অবজেক্টের মধ্যে শেয়ার করা হয়। এটি প্রতিটি অবজেক্টের জন্য পৃথক কপি না করে ক্লাসের সব অবজেক্টের জন্য একটিই কপি থাকে।

```
class MyClass:
    static_attr = 0  # Static attribute

    def __init__(self, value):
        self.value = value

# অবজেক্ট তৈরি
obj1 = MyClass(10)
obj2 = MyClass(20)

# স্ট্যাটিক অ্যাট্রিবিউট অ্যাক্সেস
print(MyClass.static_attr)  # Output: 0
print(obj1.static_attr)     # Output: 0
print(obj2.static_attr)     # Output: 0

# স্ট্যাটিক অ্যাট্রিবিউট পরিবর্তন
MyClass.static_attr = 5

print(MyClass.static_attr)  # Output: 5
print(obj1.static_attr)     # Output: 5
print(obj2.static_attr)     # Output: 5
```

স্ট্যাটিক মেথড (Static Method)

স্ট্যাটিক মেথড হল একটি মেথড যা ক্লাস বা অবজেক্টের কোনো নির্দিষ্ট ইনস্ট্যান্সের উপর নির্ভর করে না। এটি `@staticmethod` ডেকোরেটর দিয়ে সংজ্ঞায়িত করা হয় এবং সাধারণ ফাংশনের মতো কাজ করে।

```
class MyClass:
    @staticmethod
    def static_method():
        print("This is a static method.")

# স্ট্যাটিক মেথড কল করা
MyClass.static_method()  # Output: This is a static method.
```

ক্লাস মেথড (Class Method)

ক্লাস মেথড হল একটি মেথড যা ক্লাসের উপর কাজ করে এবং ক্লাসকে প্যারামিটার হিসেবে নেয়। এটি `@classmethod` ডেকোরেটর দিয়ে সংজ্ঞায়িত করা হয় এবং সাধারণত ক্লাসের স্ট্যাটিক অ্যাট্রিবিউট পরিবর্তন করতে ব্যবহৃত হয়।

```
class MyClass:
    count = 0  # Static attribute

    def __init__(self):
        MyClass.count += 1

    @classmethod
    def display_count(cls):
        print(f"Total instances: {cls.count}")

# অবজেক্ট তৈরি
obj1 = MyClass()
obj2 = MyClass()
obj3 = MyClass()

# ক্লাস মেথড কল করা
MyClass.display_count()  # Output: Total instances: 3
```

সংক্ষেপে:

1. স্ট্যাটিক অ্যাট্রিবিউট (Static Attribute):

- ক্লাসের সব অবজেক্টের মধ্যে শেয়ার করা হয়।
- উদাহরণ: `MyClass.static_attr`

1. স্ট্যাটিক মেথড (Static Method):

- ক্লাস বা অবজেক্টের নির্দিষ্ট ইনস্ট্যান্সের উপর নির্ভর করে না।
- @staticmethod ডেকোরেটর দিয়ে সংজ্ঞায়িত।
- উদাহরণ: `MyClass.static_method()`

1. ক্লাস মেথড (Class Method):

- ক্লাসের উপর কাজ করে এবং ক্লাসকে প্যারামিটার হিসেবে নেয়।

- @classmethod ডেকোরেটর দিয়ে সংজ্ঞায়িত।
- উদাহরণ: `MyClass.display_count()`

বৈশিষ্ট্য	স্ট্যাটিক মেথড	ক্লাস মেথড
ডেকোরেটর	`@staticmethod`	`@classmethod`
প্রথম প্যারামিটার	নিজস্ব কোনো ডিফল্ট প্যারামিটার নেই	cls
অ্যাক্সেস	ইনস্ট্যান্স বা ক্লাসের ডেটা অ্যাক্সেস করতে পারে না	ক্লাসের ডেটা অ্যাক্সেস ও মডিফাই করতে পারে
সম্পর্ক	ক্লাস বা ইনস্ট্যান্সের সাথে সম্পর্কিত নয়	ক্লাসের সাথে সম্পর্কিত
ব্যবহার	ইউটিলিটি ফাংশন	ক্লাস লেভেলের অপারেশন

সংক্ষেপে:

- **স্ট্যাটিক মেথড:** ক্লাস বা ইনস্ট্যান্সের নির্দিষ্ট অবস্থা বা ডেটার উপর নির্ভর করে না। ইউটিলিটি ফাংশন হিসেবে কাজ করে।
- **ক্লাস মেথড:** ক্লাসের উপর কাজ করে এবং ক্লাসের ডেটা অ্যাক্সেস ও মডিফাই করতে পারে। ক্লাস লেভেলের অপারেশন পরিচালনা করতে ব্যবহৃত হয়।

> 7.1 : Getter Setter and Read only property Using Property Decorator

Getter Setter & Read only property

Python-এ প্রোপার্টি ডেকোরেটর (@property) ব্যবহার করে আমরা গেটার, সেটার এবং রিড-অনলি প্রোপার্টি তৈরি করতে পারি। এগুলি ক্লাসের অ্যাট্রিবিউটগুলির উপর নিয়ন্ত্রণ বাড়াতে সাহায্য করে এবং ডেটা এনক্যাপসুলেশনকে সহজ করে তোলে।

গেটার এবং সেটার (Getter and Setter)

গেটার হল একটি মেথড যা একটি প্রাইভেট অ্যাট্রিবিউটের মান রিটার্ন করে এবং সেটার হল একটি মেথড যা একটি প্রাইভেট অ্যাট্রিবিউটের মান সেট করে।

```
class Person:
    def __init__(self, name):
        self._name = name  # প্রাইভেট অ্যাট্রিবিউট

    @property
    def name(self):
        print("Getting name...")
        return self._name

    @name.setter
    def name(self, value):
        print("Setting name...")
        if isinstance(value, str) and value:
            self._name = value
        else:
            raise ValueError("Name must be a non-empty
            string")

# অবজেক্ট তৈরি
person = Person("John")

# গেটার ব্যবহার করে নাম প্রিন্ট করা
print(person.name)  # Output: Getting name... John

# সেটার ব্যবহার করে নাম পরিবর্তন করা
person.name = "Jane"  # Output: Setting name...

print(person.name)  # Output: Getting name... Jane
```

রিড-অনলি প্রোপার্টি (Read-Only Property)

রিড-অনলি প্রোপার্টি এমন একটি প্রোপার্টি যা কেবলমাত্র পড়া যায়, সেট করা যায় না। এটি @property ডেকোরেটর ব্যবহার করে তৈরি করা হয়, কিন্তু কোন সেটার ডেকোরেটর সংজ্ঞায়িত করা হয় না।

```
class Circle:
    def __init__(self, radius):
        self._radius = radius  # প্রাইভেট অ্যাট্রিবিউট

    @property
    def radius(self):
        return self._radius

    @property
    def area(self):
        return 3.14159 * self._radius * self._radius

# অবজেক্ট তৈরি
circle = Circle(5)

# রিড-অনলি প্রোপার্টি ব্যবহার করে রেডিয়াস এবং এরিয়া প্রিন্ট করা
print(circle.radius)  # Output: 5
print(circle.area)    # Output: 78.53975

# রিড-অনলি প্রোপার্টি পরিবর্তনের চেষ্টা করলে এরর হবে
# circle.area = 100  # AttributeError: can't set attribute
```

সংক্ষেপে:

- **গেটার এবং সেটার** ব্যবহার করে আমরা প্রাইভেট অ্যাট্রিবিউটের উপর নিয়ন্ত্রণ বাড়াতে পারি এবং ডেটা এনক্যাপসুলেশন নিশ্চিত করতে পারি।
- **রিড-অনলি প্রোপার্টি** ব্যবহার করে আমরা প্রোপার্টির মান পরিবর্তন করার ক্ষমতা প্রতিরোধ করতে পারি, ফলে ডেটা নিরাপত্তা বাড়ে।

Inner Function & Wrapper Function

Python-এ ইননার ফাংশন এবং র‍্যাপার ফাংশনের ব্যবহার অনেক গুরুত্বপূর্ণ এবং এটি প্রোগ্রামিং এর অনেক সুবিধা প্রদান করে।

ইননার ফাংশন (Inner Function)

ইননার ফাংশন হল এমন একটি ফাংশন যা আরেকটি ফাংশনের ভিতরে সংজ্ঞায়িত হয়। ইননার ফাংশন কেবলমাত্র তার বাহিরের ফাংশনের ভিতরে কল করা যায় এবং এর বাহিরের স্কোপে থাকা ভেরিয়েবল এবং ফাংশনগুলিকে অ্যাক্সেস করতে পারে। এটি প্রোগ্রামিংয়ে ডেটা এনক্যাপসুলেশন এবং মডুলারিটি বৃদ্ধিতে সহায়ক।

```
def outer_function(text):
    def inner_function():
        print(text)
    inner_function()

outer_function("Hello, World!")  # Output: Hello, World!
```

র‍্যাপার ফাংশন (Wrapper Function)

র‍্যাপার ফাংশন হল এমন একটি ফাংশন যা আরেকটি ফাংশনকে গ্রহণ করে এবং তার উপর কিছু অতিরিক্ত কাজ সম্পাদন করে। এটি সাধারণত ডেকোরেটর ফাংশন তৈরি করতে ব্যবহৃত হয়, যেখানে মূল ফাংশনের কার্যকলাপের আগে এবং পরে অতিরিক্ত কার্যকলাপ সম্পাদন করা হয়। র‍্যাপার ফাংশন প্রোগ্রামিংয়ে কোড পুনঃব্যবহারযোগ্যতা, মডুলারিটি এবং কার্যকারিতা বাড়াতে ব্যবহৃত হয়।

```
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is
        called.")
        func()
        print("Something is happening after the function is
        called.")
    return wrapper
```

```
# @my_decorator
def say_hello():
    print("Hello!")

# say_hello()

my_decorator(say_hello())
```

> 7.3 : (optional) How does decorator work

How Decorator work

ডেকোরেটর কীভাবে কাজ করে

ডেকোরেটর Python-এর একটি শক্তিশালী বৈশিষ্ট্য যা ফাংশন বা মেথডের আচরণ পরিবর্তন করতে বা প্রসারিত করতে ব্যবহৃত হয়। এটি একটি ফাংশনকে আরেকটি ফাংশনে প্যারামিটার হিসেবে গ্রহণ করে এবং পরিবর্তিত বা প্রসারিত ফাংশনটি রিটার্ন করে।

ডেকোরেটরের কাজের মূল পদ্ধতি:

- **ডেকোরেটর ফাংশন সংজ্ঞায়িত করা:** একটি ফাংশন যা আরেকটি ফাংশনকে প্যারামিটার হিসেবে গ্রহণ করে এবং পরিবর্তিত বা প্রসারিত ফাংশন রিটার্ন করে।
- **র‍্যাপার ফাংশন সংজ্ঞায়িত করা:** ডেকোরেটর ফাংশনের ভিতরে একটি র‍্যাপার ফাংশন সংজ্ঞায়িত করা হয় যা মূল ফাংশনের আগে বা পরে অতিরিক্ত কাজ সম্পাদন করে।
- **ডেকোরেটর প্রয়োগ করা:** `@decorator_name` সিনট্যাক্স ব্যবহার করে ডেকোরেটর প্রয়োগ করা হয়।

Step: 1

```
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is
        called.")
        func()
        print("Something is happening after the function is
        called.")
    return wrapper
```

`my_decorator` নামের একটি ফাংশন সংজ্ঞায়িত করা হয়েছে যা একটি ফাংশন `func` প্যারামিটার হিসেবে গ্রহণ করে।

`wrapper` নামের একটি ইননার ফাংশন সংজ্ঞায়িত করা হয়েছে যা `func` ফাংশনের আগে এবং পরে কিছু কাজ সম্পাদন করে।

`my_decorator` ফাংশনটি `wrapper` ফাংশনটি রিটার্ন করে।

Step-2

```
def say_hello():
    print("Hello!")
```

`say_hello` নামের একটি সাধারণ ফাংশন সংজ্ঞায়িত করা হয়েছে যা `"Hello!"` প্রিন্ট করে।

Step-3

```
@my_decorator
def say_hello():
    print("Hello!")
```

`@my_decorator` ব্যবহার করে `say_hello` ফাংশনটি সজ্জিত করা হয়েছে। এটি মূলত `say_hello` ফাংশনটিকে `my_decorator` ফাংশনের মাধ্যমে র‍্যাপ করে।

সম্পূর্ণ উদাহরণ:

```
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is
        called.")
        func()
        print("Something is happening after the function is
        called.")
    return wrapper
```

```
@my_decorator
def say_hello():
    print("Hello!")

say_hello()
```

```
# Output:

Something is happening before the function is called.
Hello!
Something is happening after the function is called.
```

কিভাবে ডেকোরেটর কাজ করে:

1. ডেকোরেটর ফাংশন (`my_decorator`) একটি ফাংশন (`say_hello`) গ্রহণ করে এবং একটি নতুন ফাংশন (`wrapper`) রিটার্ন করে।
2. `@my_decorator` ব্যবহার করে `say_hello` ফাংশনটি সজ্জিত করা হয়, যা মূলত `say_hello` কে wrapper ফাংশনে রূপান্তরিত করে।
3. `say_hello()` কল করলে আসলে `wrapper` ফাংশন কল হয়, যা প্রথমে অতিরিক্ত কাজ করে, তারপর `say_hello` ফাংশন কল করে এবং শেষে আবার অতিরিক্ত কাজ করে।

আরও উদাহরণ:

লগিং ডেকোরেটর:

```
def log_decorator(func):
    def wrapper(*args, **kwargs):
        print(f"Calling function {func.__name__} with
        arguments {args} and {kwargs}”)

        result = func(*args, **kwargs)

        print(f"{func.__name__} returned {result}")
        return result
    return wrapper
```

```
@log_decorator
def add(a, b):
    return a + b

print(add(5, 3))
```

```
# Output:

Calling function add with arguments (5, 3) and {}
add returned 8
8
```

সংক্ষেপে:

1. **ডেকোরেটর:** একটি ফাংশন যা আরেকটি ফাংশনের আচরণ পরিবর্তন বা প্রসারিত করতে ব্যবহৃত হয়।
2. **র‍্যাপার ফাংশন:** ডেকোরেটর ফাংশনের ভিতরে সংজ্ঞায়িত হয় এবং মূল ফাংশনের আগে বা পরে অতিরিক্ত কাজ সম্পাদন করে।
3. **প্রয়োগ:** @decorator_name সিনট্যাক্স ব্যবহার করে ডেকোরেটর প্রয়োগ করা হয়।
4. **উদাহরণ:** লগিং, অথেনটিকেশন, ক্যাশিং ইত্যাদির জন্য ডেকোরেটর ব্যবহার করা যেতে পারে।

> 7.4 : Class Composition. inheritance vs composition

Class Composition inheritance vs Composition

ক্লাস কম্পোজিশন (Class Composition)

ক্লাস কম্পোজিশন হল একটি পদ্ধতি যেখানে একটি ক্লাস তার বৈশিষ্ট্য এবং কার্যকারিতার জন্য অন্য একটি বা একাধিক ক্লাসের অবজেক্ট ধারণ করে। এটি ইনহেরিটেন্সের একটি বিকল্প যেখানে একাধিক ক্লাসের মধ্যে কোড পুনঃব্যবহারযোগ্যতা অর্জন করা যায়।

```
class Engine:
    def start(self):
        print("Engine started")
```

```
class Wheels:
    def rotate(self):
        print("Wheels are rotating")

class Car:
    def __init__(self):
        self.engine = Engine()  # Composition
        self.wheels = Wheels()  # Composition

    def drive(self):
        self.engine.start()
        self.wheels.rotate()
        print("Car is driving")

# অবজেক্ট তৈরি এবং ফাংশন কল
my_car = Car()
my_car.drive()
```

আউটপুট:

```
Engine started
Wheels are rotating
Car is driving
```

Inheritance vs Composition

ইনহেরিটেন্স (Inheritance)

ইনহেরিটেন্স হল একটি পদ্ধতি যেখানে একটি ক্লাস অন্য একটি ক্লাস থেকে বৈশিষ্ট্য এবং মেথডগুলি উত্তরাধিকার সূত্রে পায়।

```
class Animal:
    def eat(self):
        print("Eating")

class Dog(Animal):
    def bark(self):
        print("Barking")

# অবজেক্ট তৈরি এবং ফাংশন কল
dog = Dog()
dog.eat()
dog.bark()
```

আউটপুট:

```
Eating
Barking
```

সুবিধা:
- কোড পুনঃব্যবহারযোগ্যতা: পেরেন্ট ক্লাসের বৈশিষ্ট্য এবং মেথডগুলি সাবক্লাসে ব্যবহার করা যায়।
- সম্পর্ক: একটি শক্তিশালী সম্পর্ক নির্দেশ করে যেখানে সাবক্লাস আসলে পেরেন্ট ক্লাসের একটি বিশেষায়িত রূপ।

অসুবিধা:
- শক্তিশালী সম্পর্ক: পরিবর্তন করলে সব সাবক্লাসে প্রভাব পড়তে পারে।
- কোডের জটিলতা: বড় ইনহেরিটেন্স চেইন কোডের জটিলতা বাড়াতে পারে।

কম্পোজিশন (Composition)

কম্পোজিশন হল একটি পদ্ধতি যেখানে একটি ক্লাস অন্য এক বা একাধিক ক্লাসের অবজেক্ট ধারণ করে এবং তাদের কার্যকারিতা ব্যবহার করে।

```
class Engine:
    def start(self):
        print("Engine started")

class Car:
    def __init__(self):
        self.engine = Engine()  # Composition

    def drive(self):
        self.engine.start()
        print("Car is driving")

# অবজেক্ট তৈরি এবং ফাংশন কল
my_car = Car()
my_car.drive()
```

আউটপুট:

```
Engine started
Car is driving
```

সুবিধা:

- নমনীয়তা: কম্পোজিশন আরও নমনীয় এবং পরিবর্তনশীল কোডের জন্য উপযুক্ত।
- মডুলারিটি: কোড আরও মডুলার এবং পুনঃব্যবহারযোগ্য হয়।

অসুবিধা:

- অবজেক্ট সৃষ্টি: কম্পোজিশনে অবজেক্ট তৈরি এবং পরিচালনা আরও জটিল হতে পারে।
- সম্পর্ক: ইনহেরিটেন্সের তুলনায় কম্পোজিশন সম্পর্ক কম দৃঢ়।

বৈশিষ্ট্য	**ইনহেরিটেন্স (Inheritance)**	**কম্পোজিশন (Composition)**
সংজ্ঞা	একটি ক্লাস অন্য ক্লাস থেকে উত্তরাধিকার সূত্রে পায়	একটি ক্লাস অন্য এক বা একাধিক ক্লাসের অবজেক্ট ধারণ করে
ব্যবহার	"is-a" সম্পর্ক নির্দেশ করে	"has-a" সম্পর্ক নির্দেশ করে
কোড পুনঃব্যবহার	পেরেন্ট ক্লাসের বৈশিষ্ট্য এবং মেথড ব্যবহার করে	কম্পোজিশনের মাধ্যমে কোড পুনঃব্যবহার করা যায়

নমনীয়তা	কম নমনীয়	আরও নমনীয়
মডুলারিটি	কম মডুলার	আরও মডুলার
জটিলতা	ইনহেরিটেন্স চেইন বড় হলে জটিলতা বাড়ে	অবজেক্ট সৃষ্টি এবং পরিচালনা জটিল হতে পারে

সংক্ষেপে:
- ইনহেরিটেন্স: যখন একটি ক্লাস অন্য একটি ক্লাস থেকে বৈশিষ্ট্য এবং মেথড উত্তরাধিকার সূত্রে পায়। এটি "is-a" সম্পর্ক নির্দেশ করে।
- কম্পোজিশন: যখন একটি ক্লাস অন্য একটি বা একাধিক ক্লাসের অবজেক্ট ধারণ করে এবং তাদের কার্যকারিতা ব্যবহার করে। এটি "has-a" সম্পর্ক নির্দেশ করে।

> 7.5 : A short overview of UML Diagrams

UML Diagram

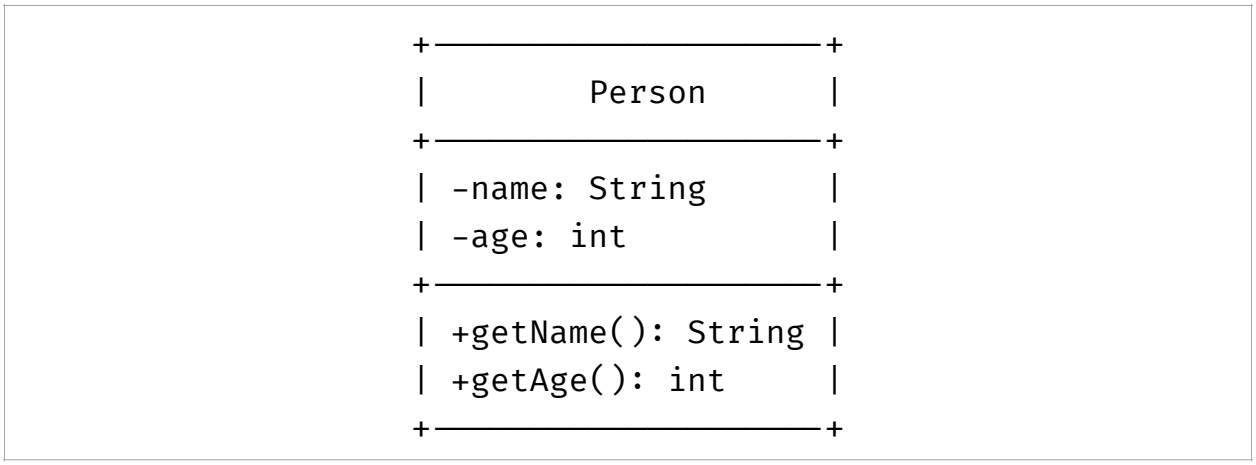
UML (Unified Modeling Language) হল একটি স্ট্যান্ডার্ডাইজড ভাষা যা সফটওয়্যার সিস্টেমগুলির ডিজাইন এবং ডকুমেন্টেশন করার জন্য ব্যবহৃত হয়। UML ডায়াগ্রামগুলি সফটওয়্যার সিস্টেমের বিভিন্ন দিকগুলি দেখায় এবং বুঝতে সাহায্য করে। এটি প্রোগ্রামার, ডিজাইনার এবং স্টেকহোল্ডারদের মধ্যে যোগাযোগ সহজ করে তোলে।

UML ডায়াগ্রামের প্রধান প্রকারভেদ

UML ডায়াগ্রামগুলি প্রধানত দুটি ক্যাটেগরিতে বিভক্ত: স্ট্রাকচারাল (Structural) ডায়াগ্রাম এবং বিহেভিয়োরাল (Behavioral) ডায়াগ্রাম।

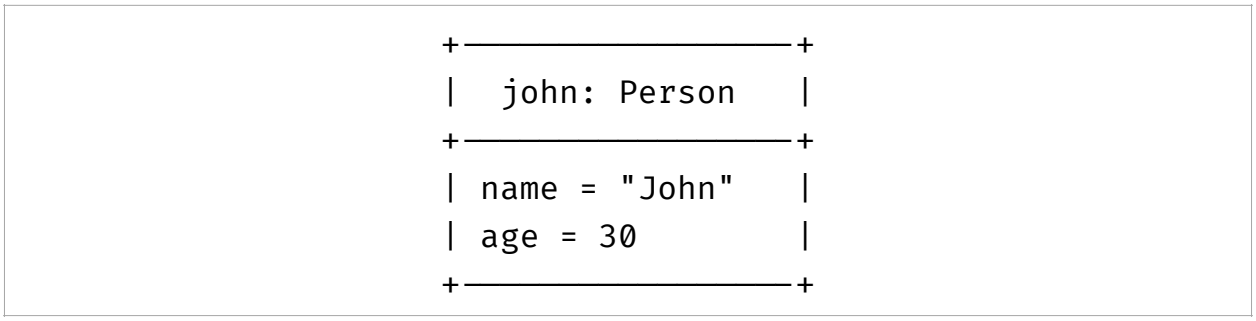
1. স্ট্রাকচারাল ডায়াগ্রাম (Structural Diagrams)

ক্লাস ডায়াগ্রাম (Class Diagram):
- সিস্টেমের ক্লাস এবং তাদের সম্পর্ক প্রদর্শন করে।
- এটি একটি বেসিক এবং সবচেয়ে সাধারণ UML ডায়াগ্রাম।
- উদাহরণ:



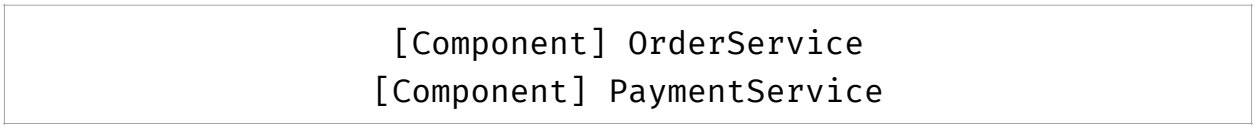
অবজেক্ট ডায়াগ্রাম (Object Diagram):

- ক্লাস ডায়াগ্রামের মতো, কিন্তু এটি একটি নির্দিষ্ট সময়ে অবজেক্টগুলির অবস্থা দেখায়।
- উদাহরণ:



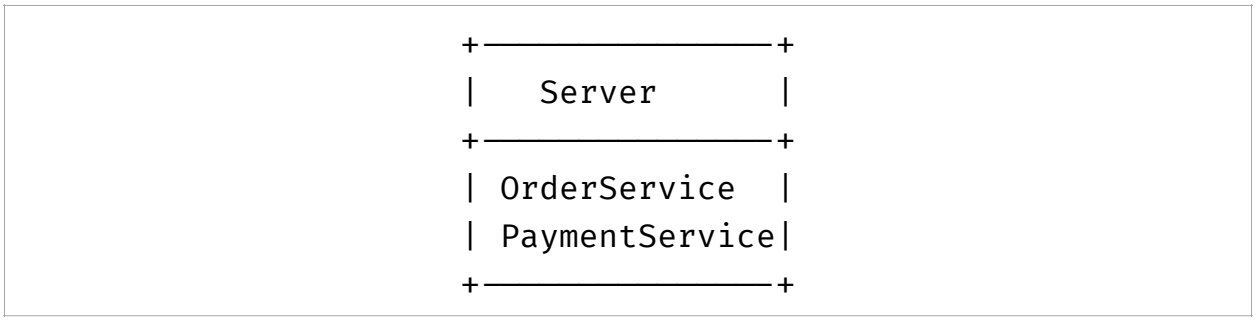
কম্পোনেন্ট ডায়াগ্রাম (Component Diagram):

- সিস্টেমের বিভিন্ন সফটওয়্যার কম্পোনেন্ট এবং তাদের সম্পর্ক দেখায়।
- উদাহরণ:



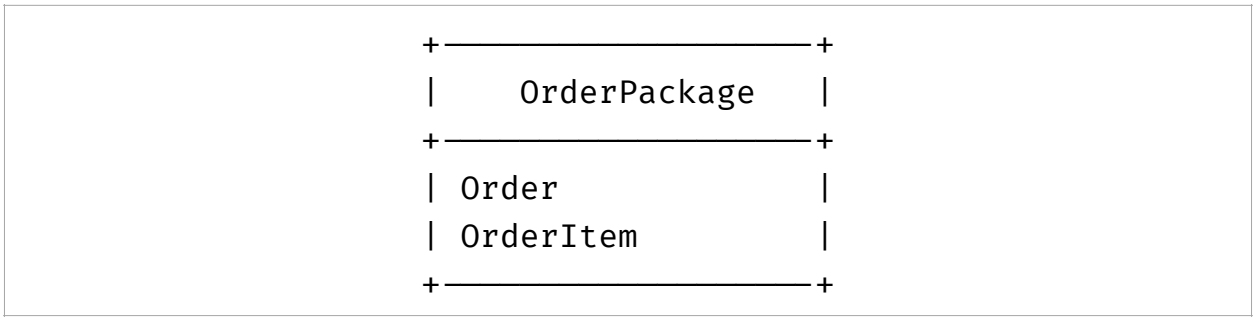
ডিপ্লয়মেন্ট ডায়াগ্রাম (Deployment Diagram):

- সিস্টেমের হার্ডওয়্যার এবং সফটওয়্যার কম্পোনেন্টগুলির ভৌত বিন্যাস প্রদর্শন করে।
- উদাহরণ:



প্যাকেজ ডায়াগ্রাম (Package Diagram):

- প্যাকেজগুলির মধ্যে সম্পর্ক এবং তাদের কন্টেন্ট দেখায়।
- উদাহরণ:



2. বিহেভিয়োরাল ডায়াগ্রাম (Behavioral Diagrams)

ইউজ কেস ডায়াগ্রাম (Use Case Diagram):

- সিস্টেমের বিভিন্ন অ্যাক্টর এবং তাদের মিথস্ক্রিয়া প্রদর্শন করে।
- উদাহরণ:

```
Actor: Customer
Use Case: Place Order
```

সিকোয়েন্স ডায়াগ্রাম (Sequence Diagram):

- বিভিন্ন অবজেক্টের মধ্যে মেসেজ পাসিংয়ের ক্রম প্রদর্শন করে।
- উদাহরণ:

```
Customer → OrderService: placeOrder()
OrderService → PaymentService: processPayment()
```

কমিউনিকেশন ডায়াগ্রাম (Communication Diagram):

- বিভিন্ন অবজেক্টের মধ্যে মেসেজ পাসিং দেখায়, কিন্তু এটি বস্তুগুলির বিন্যাসে বেশি গুরুত্ব দেয়।

• উদাহরণ:

```
Customer → OrderService: placeOrder()
OrderService → PaymentService: processPayment()
```

স্টেট ডায়াগ্রাম (State Diagram):
• একটি অবজেক্টের বিভিন্ন অবস্থার পরিবর্তন দেখায়।
• উদাহরণ:

```
[Order Placed] ⟶ [Order Shipped]
```

অ্যাক্টিভিটি ডায়াগ্রাম (Activity Diagram):
• সিস্টেমের বিভিন্ন কার্যকলাপ এবং তাদের ক্রম প্রদর্শন করে।
• উদাহরণ:

```
[Start] ⟶ [Place Order] ⟶ [Process Payment] ⟶ [End]
```

টাইমিং ডায়াগ্রাম (Timing Diagram):
• একটি নির্দিষ্ট সময়ে অবজেক্টের অবস্থা পরিবর্তন দেখায়।
• উদাহরণ:

```
State 1 --(time)⟶ State 2
```

ইন্টারেকশন ওভারভিউ ডায়াগ্রাম (Interaction Overview Diagram):
• সিকোয়েন্স এবং অ্যাক্টিভিটি ডায়াগ্রামের সমন্বয় প্রদর্শন করে।
• উদাহরণ:

```
[Sequence 1] ⟶ [Activity 1] ⟶ [Sequence 2]
```

UML ডায়াগ্রামগুলির সুবিধা:

1. **ভিজ্যুয়াল রিপ্রেজেন্টেশন:** সফটওয়্যার সিস্টেমের জটিলতা হ্রাস করে এবং বোঝার সহজ করে তোলে।
2. **ডকুমেন্টেশন:** সিস্টেমের বিভিন্ন অংশের ডকুমেন্টেশন তৈরি করতে সাহায্য করে।
3. ***যোগাযোগ:*** ডেভেলপার, ডিজাইনার এবং স্টেকহোল্ডারদের মধ্যে পরিষ্কারভাবে ধারণা বিনিময় করতে সাহায্য করে।
4. **ডিজাইন:** সিস্টেম ডিজাইন এবং স্থাপনা পরিকল্পনা করতে সাহায্য করে।
5. **প্রব্লেম সলভিং:** সিস্টেমের সমস্যাগুলি শনাক্ত করতে এবং তাদের সমাধান করতে সহায়তা করে।

সংক্ষেপে:

1. **স্ট্রাকচারাল ডায়াগ্রাম:** ক্লাস, অবজেক্ট, কম্পোনেন্ট, ডিপ্লয়মেন্ট এবং প্যাকেজ ডায়াগ্রাম অন্তর্ভুক্ত।
2. **বিহেভিয়োরাল ডায়াগ্রাম:** ইউজ কেস, সিকোয়েন্স, কমিউনিকেশন, স্টেট, অ্যাক্টিভিটি, টাইমিং এবং ইন্টারেকশন ওভারভিউ ডায়াগ্রাম অন্তর্ভুক্ত।

> 7.5 : Design Patterns Singleton, Factory, Builder, etc

Design Patterns

ডিজাইন প্যাটার্নস হল প্রমাণিত সমাধান যা সাধারণ সফটওয়্যার ডিজাইন সমস্যার সমাধান করতে ব্যবহৃত হয়। এগুলি প্রোগ্রামিংয়ের শ্রেষ্ঠ অনুশীলনগুলি সংজ্ঞায়িত করে যা পুনঃব্যবহারযোগ্য, নমনীয় এবং কার্যকর কোড লিখতে সহায়তা করে। এখানে কয়েকটি সাধারণ ডিজাইন প্যাটার্ন নিয়ে আলোচনা করা হয়েছে: সিঙ্গলটন, ফ্যাক্টরি, বিল্ডার, ইত্যাদি।

সিঙ্গলটন প্যাটার্ন (Singleton Pattern)

সিঙ্গলটন প্যাটার্ন একটি প্যাটার্ন যেখানে একটি ক্লাসের কেবলমাত্র একটি ইনস্ট্যান্স তৈরি করা যায়। এটি গ্লোবাল অ্যাক্সেস পয়েন্ট প্রদান করে এবং নিশ্চিত করে যে শুধুমাত্র একটি অবজেক্ট তৈরি হয়।

```
class Singleton:
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super(Singleton, cls).__new__(cls)
        return cls._instance

# অবজেক্ট তৈরি
obj1 = Singleton()
obj2 = Singleton()

print(obj1 is obj2)  # Output: True
```

ফ্যাক্টরি প্যাটার্ন (Factory Pattern)

ফ্যাক্টরি প্যাটার্ন একটি প্যাটার্ন যেখানে একটি ইন্টারফেস বা অ্যাবস্ট্রাক্ট ক্লাস ব্যবহার করে অবজেক্ট তৈরি করা হয়। এটি ক্লায়েন্ট কোড থেকে অবজেক্ট তৈরির প্রক্রিয়াটি আলাদা করে।

```
class Dog:
    def speak(self):
        return "Woof!"

class Cat:
    def speak(self):
        return "Meow!"

class AnimalFactory:
    @staticmethod
    def get_animal(animal_type):
        if animal_type == "dog":
            return Dog()
        elif animal_type == "cat":
            return Cat()
        return None

# ফ্যাক্টরি ব্যবহার করে অবজেক্ট তৈরি
animal = AnimalFactory.get_animal("dog")
print(animal.speak())  # Output: Woof!
```

বিল্ডার প্যাটার্ন (Builder Pattern)

বিল্ডার প্যাটার্ন একটি প্যাটার্ন যা একটি জটিল অবজেক্টের নির্মাণ প্রক্রিয়া পৃথক এবং স্বাধীন উপাদানগুলিতে ভাগ করে। এটি ধাপে ধাপে একটি অবজেক্ট তৈরি করতে সাহায্য করে।

```
class Car:
    def __init__(self):
        self.wheels = None
        self.engine = None
        self.body = None
```

```
    def __str__(self):
        return f"Car with {self.wheels} wheels, {self.engine}
        engine, and {self.body} body."

class CarBuilder:
    def __init__(self):
        self.car = Car()

    def add_wheels(self, wheels):
        self.car.wheels = wheels
        return self

    def add_engine(self, engine):
        self.car.engine = engine
        return self

    def add_body(self, body):
        self.car.body = body
        return self

    def build(self):
        return self.car

# বিল্ডার ব্যবহার করে কার তৈরি
builder = CarBuilder()
car =
builder.add_wheels("4").add_engine("V8").add_body("SUV").build
()
print(car)  # Output: Car with 4 wheels, V8 engine, and SUV
body.
```

পাইথন OOP প্রজেক্টস - Restaurant management

প্রজেক্ট ইন্ট্রোডাকশন :

এই মডিউলে আমরা OOP কনসেপ্টস ব্যবহার করে ইন্টারেস্টিং একটি কনসোল প্রজেক্ট কওরে ফেলব ।

প্রজেক্টটির নাম দেয়া যেতে পারে - Restaurant Management

প্রজেক্টের ফিচারস ও স্পেসিফিকেশন গুলো দেখে নেই-

Introduction	The Restaurant Management System is a Python-based terminal application designed to streamline restaurant operations, including menu management, order processing, employee management, and customer interactions. The system allows both customers and administrators to perform various tasks efficiently, contributing to a seamless dining experience.
Customer Features	**View Menu:** Customers can browse through the restaurant's menu to explore available food and beverage options. **Add Item to Cart:** Customers can add desired items to their shopping cart for ordering. **View Cart:** Customers can view the contents of their cart, including item names, quantities, and prices. **Pay Bill:** Customers can pay their bill, completing the transaction and clearing the cart. **Account Creation:** Customers have the option to create an account with their name, phone number, email, and address for personalized services.

Admin Features	**Add New Item:** Admins can add new items to the restaurant's menu, specifying details such as name, price, and quantity. **Add Employee:** Admins can add new employees to the system, including details like name, phone number, email, address, age, designation, and salary. **View Employee List:** Admins can view a list of all employees along with their details. **View Items:** Admins can view the restaurant's menu, displaying available items along with their prices and quantities. **Delete Item:** Admins can remove items from the menu, providing flexibility in menu management.

User ক্লাস তৈরি :

RestaurantManagement নামে প্রজেক্ট ফোল্ডার ক্রিয়েট কওরে সেখানে `users.py` ফাইল ক্রিয়েট করি-

এখন, নিচের মত কওরে User ক্লাস টি লিখে ফেলি- `users.py`

```
from abc import ABC
from orders import Order
class User(ABC):
    def __init__(self, name, phone, email, address):
        self.name = name
        self.email = email
        self.address = address
        self.phone = phone
```

!! ABC ক্লাস কে ইনহেরিট করার মাধ্যমে User ক্লাস কে Abstract ক্লাস হিসেবে তৈরি করা হয়েছে

User ক্লাস কে ইনহেরিট কওরে Employee ক্লাস লিখে ফেলি- users.py

```
class Employee(User):
    def __init__(self, name, email, phone, address, age,
designation, salary):
        super().__init__(name, phone, email, address)
        self.age = age
        self.designation = designation
        self.salary = salary
```

অনুরুপ ভাবে Admin ক্লাস তৈরি কওরে এতে **add_employee( ) , view_employee( )** নামের ২ টি মেথড এ্যাড কওরে ফেলি- users.py

```
class Admin(User):
    def __init__(self, name, email, phone, address):
        super().__init__(name, phone, email, address)

    def add_employee(self, restaurent, employee):
        restaurent.add_employee(employee)

    def view_employee(self, restaurent):
        restaurent.view_employee()
```

Menu ও Restaurant ক্লাস তৈরি :

এই মডিউলে আমরা Restaurant ও Menu ক্লাস তৈরি করব।

Menu ক্লাস লিখার জন্য menu.py নামে একটি পাইথন ফাইল ক্রিয়েট কওরে ফেলি।

menu.py ফাইলে Menu ক্লাস টি লিখে ফেলি- menu.py

```
class Menu:
    def __init__(self):
        self.items = []  # items er database

    def add_menu_item(self, item):
        self.items.append(item)
```

```
    def find_item(self, item_name):
        for item in self.items:
            if item.name.lower() == item_name.lower():
                return item
        return None

    def remove_item(self, item_name):
        item = self.find_item(item_name)
        if item:
            self.items.remove(item)
            print("Item deleted")
        else:
            print("item not found")

    def show_menu(self):
        print("*****Menu*****")
        print("Name\tPrice\tQuantity")
        for item in self.items:
            print(f"{item.name}\t{item.price}
            \t{item.quantity}")
```

এখন , `restaurant.py` নামে একটি পাইথন ফাইল ক্রিয়েট কওরে ফেলি।

`restaurant.py` ফাইলে Restaurent ক্লাস টী লিখে ফেলি- `restaurent.py`

```
from menu import Menu

class Restaurent:
    def __init__(self, name):
        self.name = name
        self.employees = []  # eta hocche amader database
        self.menu = Menu()

    def add_employee(self, employee):
        self.employees.append(employee)
```

```
    def view_employee(self):
        print("Employee List!!")
        for emp in self.employees:
            print(emp.name, emp.email, emp.phone, emp.address)
```

FoodItem ক্লাস :

এই মডিউলে আমরা FoodItem ক্লাস তৈরি করব।

FoodItem ক্লাস লিখার জন্য `food_item.py` নামে একটি পাইথন ফাইল ক্রিয়েট কওরে ফেলি।

`food_item.py` ফাইলে FoodItem ক্লাস টি লিখে ফেলি- `food_item.py`

```
class FoodItem:
    def __init__(self, name, price, quantity):
        self.name = name
        self.price = price
        self.quantity = quantity
```

এখন মেনু তে নতুন আইটেম এ্যাড, আইটেম ডিলিট, মেনু দেখার জন্য Admin ক্লাসে নিচের মেথড গুলো লিখে ফেলি- `users.py`

```
class Admin(User):

    ...
    ...
    ...

    def add_new_item(self, restaurent, item):
        restaurent.menu.add_menu_item(item)

    def remove_item(self, restaurent, item):
        restaurent.menu.remove_item(item)

    def view_menu(self, retaurent):
        retaurent.menu.show_menu()
```

Customer ক্লাস তৈরি ও Cart ম্যানেজমেন্ট :

এই মডিউলে আমরা Customer ক্লাসটি তৈরি করব। - `users.py`

```
...
...
...

class Customer(User):
    def __init__(self, name, email, phone, address):
        super().__init__(name, phone, email, address)
        self.cart = Order()

    def view_menu(self, restaurent):
        restaurent.menu.show_menu()

    def add_to_cart(self, restaurent, item_name, quantity):
        item = restaurent.menu.find_item(item_name)
        if item:
            if quantity > item.quantity:
                print("Item quantity exceeded!!")
            else:
                item.quantity = quantity
                self.cart.add_item(item)
                print("item added")
        else:
            print("Item not found”)

    def view_cart(self):
        print("**View Cart**")
        print("Name\tPrice\tQuantity")
        for item, quantity in self.cart.items.items():
            print(f"{item.name}\t{item.price}\t{quantity}")
        print(f"Total Price : {self.cart.total_price}")
```

```
   def pay_bill(self):
        print(f"Total {self.cart.total_price} paid
        successfully")
        self.cart.clear()

    ...
    ...
    ...
```

Order ক্লাস :

এই মডিউলে আমরা দেখব কাস্টমারের অর্ডার গুলো কিভাবে ম্যানেজ করা যায়।

এর জন্য oders.py ফাইল ক্রিয়েট কওরে সেখানে Order নামের ক্লাসটি নিচের মত কওরে লিখে ফেলি- orders.py

```
class Order:
    def __init__(self) → None:
        self.items = {}

    def add_item(self, item):
        if item in self.items:
            # jodi item ta cart e already thake
            self.items[item] += item.quantity
        else:
            self.items[item] = item.quantity
           # cart e item jodi na thake

    def remove(self, item):
        if item in self.items:
            del self.items[item]

    @property
```

```
    def total_price(self):
        return sum(item.price * quantity for item, quantity in
self.items.items())

    def clear(self):
        self.items = {}
```

!! total_price( ) মেথডে @property ডেকোরেটর ব্যবহার করা হয়েছে ।

কাস্টমার ইন্টারফেস :

এই মডিউলে আমরা কাস্টমারের জন্য ইন্টারফেস তৈরি করব।

অর্থ্যাৎ, প্রগ্রামটি রান করা হলে যেন কাস্টমার কনসোলে তার জন্য এভেইলেবল অপশনগুলো দেখতে পারে এবং নেভিগেট করতে পারে।

এর জন্য main.py নামে একটি পাইথন ফাইল ক্রিয়েট কওরে ফেলি।

```
from food_item import FoodItem
from menu import Menu
from users import Customer, Admin, Employee
from restaurent import Restaurent
from orders import Order

mamar_restaurent = Restaurent("Mamar Restaurement”)

def customer_menu():
    name = input("Enter Your Name : ")
    email = input("Enter Your Email : ")
    phone = input("Enter Your Phone : ")
    address = input("Enter Your Address : ")
    customer = Customer(name=name, email=email, phone=phone,
    address=address)
```

```
while True:
    print(f"Welcome {customer.name}!!")
    print("1. View Menu")
    print("2. Add item to cart")
    print("3. View Cart")
    print("4. PayBill")
    print("5. Exit")

    choice = int(input("Enter Your Choice : "))
    if choice == 1:
        customer.view_menu(mamar_restaurent)
    elif choice == 2:
        item_name = input("Enter item name : ")
        item_quantity = int(input("Enter item quantity :
        “))

        customer.add_to_cart(mamar_restaurent, item_name,
        item_quantity)

    elif choice == 3:
        customer.view_cart()
    elif choice == 4:
        customer.pay_bill()
    elif choice == 5:
        break
    else:
        print("Invalid Input")
```

!! দেখা যাচ্ছে, ফাইলের শুরুতেই প্রয়োজনীয় মডিউলগুলো (যেসব ক্লাস, মেথডস আমরা আগের মডিউল গুলোতে তৈরি করেছি) ইমপোর্ট কওরে নেয়া হয়েছে।

এরপর mamar_restaurent নামে একটি Restaurent অবজেক্ট তৈরি করা হয়েছে।

customer_menu( ) নামে ফাংশন তৈরি কওরে সেখানে কাস্টমারের জন্য রিলেভেন্ট অপশন সমূহ রাখা হয়েছে এবং ম্যানেজ করা হয়েছে।

এডমিন ইন্টারফেস :

এই মডিউলে আমরা এডমিনের জন্য ইন্টারফেস তৈরি করব।

অর্থ্যাৎ, প্রগ্রামটি রান করা হলে যেন েডমিন কনসোলে তার জন্য এভেইলেবল অপশনগুলো দেখতে পারে এবং নেভিগেট করতে পারে।

এর জন্য main.py ফাইলে admin_menu( ) নামে একটি ফাংশন লিখে ফেলি-main.py

```
...
...
...


def admin_menu():
    name = input("Enter Your Name : ")
    email = input("Enter Your Email : ")
    phone = input("Enter Your Phone : ")
    address = input("Enter Your Address : ")
    admin = Admin(name=name, email=email, phone=phone,
    address=address)

    while True:
        print(f"Welcome {admin.name}!!")
        print("1. Add New Item")
        print("2. Add New Employee")
        print("3. View Employee")
        print("4. View Items")
        print("5. Delete Item")
        print("6. Exit")

        choice = int(input("Enter Your Choice : "))
```

```
        if choice == 1:
            item_name = input("Enter Item Name : ")
            item_price = int(input("Enter Item Price : "))
            item_quantity = int(input("Enter Item Quantity :
            "))
            item = FoodItem(item_name, item_price,
            item_quantity)
            admin.add_new_item(mamar_restaurent, item)

        elif choice == 2:
            name = input("Enter employee name : ")
            phone = input("Enter employee phone : ")
            email = input("Enter employee email : ")
            designation = input("Enter employee designation :
            ")
            age = input("Enter employee age : ")
            salary = input("Enter employee salary : ")
            address = input("Enter employee address : ")
            employee = Employee(name, email, phone, address,
                                age, designation, salary)

            admin.add_employee(mamar_restaurent, employee)
        elif choice == 3:
            admin.view_employee(mamar_restaurent)
        elif choice == 4:
            admin.view_menu(mamar_restaurent)
        elif choice == 5:
            item_name = input("Enter item name : ")
            admin.remove_item(mamar_restaurent, item_name)
        elif choice == 6:
            break
        else:
            print("Invalid Input")
```

!! admin_menu( ) নামে ফাংশন তৈরি কওরে সেখানে এডমিনের জন্য রিলেভেন্ট অপশন সমূহ রাখা হয়েছে এবং ম্যানেজ করা হয়েছে।

রানিং দ্যা সিস্টেম :

এই মডিউলে আমরা main.py ফাইল থেকে সিস্টেম টি রান করার ব্যবস্থা করব ।

এর জন্য main.py ফাইলে কন্ডিশন True রেখে নিচের মত করে while লুপ লিখে ফেলি-

main.py

```
...
...
...

while True:
    print("Welcome!!")
    print("1. Customer")
    print("2. Admin")
    print("3. Exit")
    choice = int(input("Enter your choice : "))
    if choice == 1:
        customer_menu()
    elif choice == 2:
        admin_menu()
    elif choice == 3:
        break
    else:
        print("Invalid Input!!")
```

এবার main.py ফাইলটি টার্মিনালে রান করলে আমরা সিস্টেম টি নেভিগেট করতে পারব-

```
# output:

Welcome!!
1. Customer
2. Admin
3. Exit

Enter your choice:
```

Ride Sharing

এই মডিউল এ আমরা একটা রাইড শেয়ারিং প্রজেক্ট তৈরি করব-

Creating Rider class :

শুরুতে আমরা `users.py` নামে একটা ফাইল খুলে ফেলি তারপর সেখানে একটা `Abstract Class` তৈরি করে User নামে-

```
from abc import ABC, abstractmethod

class User(ABC):
    def __init__(self, name, email, nid) → None:
        self.name = name
        self.email = email
        self.nid = nid
        self.wallet = 0

    @abstractmethod
    def display_profile(self):
        raise NotImplementedError
```

এখানে আমরা `display_profile` নামে একটা `abstract_method` ও তৈরি করছি এবং সেটা `NotImplementedError` নামে একটা ইরর রেইজ করবে যদি User ক্লাস এর কোনো সাবক্লাস এই মেথডটিকে ইমপ্লিমেন্ট না করে।
এরপর আমরা Rider নামে একটা ক্লাস তৈরি করব যেখানে User ক্লাসটিকে ইনহেরিট করা হবে এবং Rider ক্লাস এর `__init__` মেথড এর মধ্যে আমরা super() কন্সট্রাক্টর কে কল করে ইউসার ক্লাস এর জন্য প্রয়োজনীয় প্যারামিটার পাঠিয়ে দিবো

```
class Rider(User):
    def __init__(self, name, email, nid, current_location,
    initial_amount) → None:
        super().__init__(name, email, nid)
        self.current_ride = None
        self.wallet = initial_amount
        self.current_location = current_location
```

```
    def display_profile(self):
        print(f"Rider : {self.name} and email {self.email}")

    def load_cash(self, amount):
        if amount > 0:
            self.wallet += amount
        else:
            print("Amount less than 0")

    def update_location(self, current_location):
        self.current_location = current_location
```

এইখানে আমরা display_profile() নামে একটা মেথড ডিক্লেয়ার করেছি যেটা ইউজার এর ডিটেইলসগুলোকে একটা স্ট্রিং হিসেবে প্রিন্ট করবে

load_cash() নামে একটা মেথড ডিক্লেয়ার করা হয়েছে যেটার কাজ হচ্ছে যদি রাইডারের ওয়ালেট ব্যালেন্স ০ এর থেকে বেশি হয় তাহলে সেটার সাথে এমাউন্টটা যোগ করা আর যদি এটা ০ এর থেকে কম হয় তাহলে একটা ইরর মেসেজ প্রিন্ট করবে Amount less than 0

Driver Class :

আমরা গত মডিউল এ Rider ক্লাস তৈরি করেছিলাম সেখানে আমরা আরো ২ টি মেথড ইম্পপ্লিমেন্ট করব request_ride() এবং show_current_ride() নামে।

```
class Rider(User):
    ......
    ......
    ......
    def request_ride(self, ride_sharing, destination,
vehicle_type):
        pass

    def show_current_ride(self):
        print(self.current_ride)
```

এই মেথডগুলোকে আমরা আপাতত ডিফাইন করে রাখছি সামনে আমরা এইগুলো ইমপ্লিমেন্ট করব।

এখন আমরা একইভাবে একটা Driver ক্লাস তৈরি করব

```
class Driver(User):
    def __init__(self, name, email, nid, current_location):
        super().__init__(name, email, nid)
        self.current_location = current_location
        self.wallet = 0

    def display_profile(self):
        print(f"Driver Name : {self.name}")

    def accept_ride(self, ride):
        ride.start_ride()
        ride.set_driver(self) # driver er object

    def reach_destination(self, ride):
        ride.end_ride()
```

এখানে আমরা **__init__** মেথড এর মধ্যে Driver এর কারেন্ট লোকেশন রাখতে পারে। **display_profile()** নামে একটা মেথড ক্রিয়েট করেছি ড্রাইভারের ইনফোগুলো দেখার জন্য

accept_ride() নামে একটা মেথড তৈরি করেছি যেখানে কোনো রাইডার যদি রাইড রিকুয়েস্ট দেয় সেটা যেন ড্রাইভার এক্সেপ্ট করতে পারে।

Ride Class :

এইবার আমরা `ride.py` নামে আমরা একটা ক্লাস তৈরি করব যেখানে আমরা Ride এর জন্য প্রয়োজনিয় ফাংশনালিটি ইমপ্লিমেন্ট করব

শুরুতে আমরা Ride নামে একটা ক্লাস তৈরি করি-

```
class Ride:
    def __init__(self, start_location, end_location, vehicle)
    → None:
        self.start_location= start_location
        self.end_location = end_location
        self.driver = None
        self.rider = None
        self.start_time = None
        self.end_time = None
        self.estimated_fare =
        self.calculate_fare(vehicle.vehicle_type)

        self.vehicle = vehicle

    def set_driver(self, driver):
        self.driver = driver

    def start_ride(self):
        self.start_time = datetime.now()
    def end_ride(self):
        self.end_time = datetime.now()
        self.rider.wallet -= self.estimated_fare
        self.driver.wallet += self.estimated_fare

   def __repr__(self):
        return f"Ride details. Started {self.start_location}
        to {self.end_location}”
```

এখানে আমরা Ride ক্লাস এর জন্য প্রয়োজনীয় এট্রিবিউট ডিক্লেয়ার করছি তারপর ৩ টা মেথড তৈরি করেছি। set_driver() রাইড এর জন্য ড্রাইভার সেট করার জন্য। start_ride() রাইড যখন শুরু হবে সেই সময়টা রেকর্ড করার জন্য। end_ride() রাইড শেষ হওয়ার পরে প্রয়োজনীয় কাজ যেমন end_time রেকর্ড করা, রাইডারের ওয়ালেট থেকে ব্যালেন্স কেটে নিয়ে তা ড্রাইভার এর ওয়ালেট এ যোগ করা হয়েছে। এরপর আমরা একটা __repr__ মেথড তৈরি করছি ক্লাসটিকে প্রিন্ট করে যাতে করে সেটা স্ট্রিং হিসেবে রিপ্রেজেন্ট হয়।

Ride Request :

এইবার আমরা রাইড এর রিকুয়েস্ট ম্যানেজ করার জন্য RideRequest নামে একটা ক্লাস তৈরি করব যেখানে আমরা রাইডার এর অব্জেক্ট এবং তার গন্তব্য বা এন্ড লোকেশন রাখব —

```
class RideRequest:
    def __init__(self, rider, end_location) → None:
        self.rider = rider
        self.end_location = end_location
```

এখন আমাদের জন্য আরেকটি প্রয়োজনীয় কাজ হচ্ছে রাইড এর জন্য ড্রাইভার ফাইন্ড করা তার জন্য আমরা আরেকটি ক্লাস তৈরি করব RideMatching নামে—

```
class RideMatching:
    def __init__(self, drivers) → None:
        self.available_drivers = drivers

    def find_driver(self, ride_request, vehicle_type):
        if len(self.available_drivers) > 0:
            print("Looking for drivers.....")
            driver = self.available_drivers[0]

            if vehicle_type == 'car':
                vechicle = Car('car', "ABC456", 30)
            elif vehicle_type == 'bike':
                vechicle = Bike("bike", "1234BH", 50)
            ride = Ride(ride_request.rider.current_location,
                        ride_request.end_location, vechicle)
            driver.accept_ride(ride)
            return ride
```

এই ক্লাস এর মধ্যে আমরা **find_driver()** নামে একটা মেথড তৈরি করেছি যেখানে শুরুতে আমাদের যতজন ড্রাইভার এভেইলেবল আছে তাদেরকে driver নামে এক্টা এট্রিবিউট এর মধ্যে রাখা হয়েছে এবং রাইডার এর সিলেক্ট করা **vehicle_type** এর উপর ভিত্তি করে একটা vehicle ক্লাস অব্জেক্ট তৈরিকরা হয়েছে। এখন এই **vehicle, rider.current_location** এবং **rider.end_location()** দিয়ে আমরা একটা রাইড তৈরি করব যেটা কিনা Ride ক্লাস যেটা আমরা তৈরি করেছিলাম তার ইন্সট্যান্স। এরপর আমরা ড্রাইভার এর রিকুয়েস্ট টি এক্সেপ্ট করার জন্য একটা **accept_ride()** মেথড বানাবো Driver ক্লাস এর মধ্যে যেটাকে এই **find_driver** মেথড এর মধ্যে কল করা হয়েছে এবং প্যারামিটার হিসেবে পাঠিয়ে দেওয়া হয়েছে আমাদের ride instance

```
class Driver(User):
       .....
       .....
       .....
         def accept_ride(self, ride):
                ride.start_ride()
                ride.set_driver(self) # driver er object
```

এখানে আমরা **accept_ride** কে এমনভাবে ইমপ্লিমেন্ট করছি যাতে করে ড্রাইভার এটাকে এক্সেপ্ট করলে ride যেটা প্যারামিটার হিসেবে পাঠিয়েছিলাম তার **start_method()** কে কল করে রাইড স্টার্ট করবে। এবং Ride এর জন্য Driver সেট হয়ে যাবে যেটাকে **ride.set_driver(self)** এইভাবে ইমপ্লিমেন্ট করা হয়েছে। এখানে মনে রাখতে হবে যেই self টি পাঠানো হচ্ছে সেটা মুলত এই ড্রাইভার ক্লাস এর অব্জেক্ট টা নিজেই যে **set_driver()** কে কল করছে। এর মাধ্যমে একটা ড্রাইভার ইন্সট্যান্স প্যারামিটার আকারে সেট ড্রাইভার মেথডে পাস হবে যেটা রাইড এর জন্য ড্রাইভারকে এসাইন করে দিবে।

RideSharing and Vehicle class :

এই মডিউল এ আমরা ভেহিকল নিয়ে কাজ করব। তারজন্য শুরুতে আমরা Vehicle নামে একটা `Abstract` ক্লাস ক্রিয়েট করে ফেলব তার জন্য শুরুতে `vehicles.py` নামে একটা ফাইল খুলি

```
from abc import ABC

lass Vehicle(ABC):
    speed = {
        'car' : 50,
        'bike' : 60,
        'cng' : 15
    }
    def __init__(self,vehicle_type, license_plate, rate) →
None:
        self.vehicle_type = vehicle_type
        self.license_plate = license_plate
        self.rate = rate
```

এখন এই ভেহিকল ক্লাস কে ইনহেরিট করে Car এবং Bike নামে ২ টা ক্লাস তৈরি করি—

```
class Car(Vehicle):
    def __init__(self, vehicle_type, license_plate, rate) →
    None:
        super().__init__(vehicle_type, license_plate, rate)

class Bike(Vehicle):
    def __init__(self, vehicle_type, license_plate, rate) →
    None:
        super().__init__(vehicle_type, license_plate, rate)
```

এখানে আমরা কোনো এক্সট্রা কাজ না করে ডিরেক্ট প্যারেন্ট ক্লাস এর কন্সট্রাক্টরকে কল করে দিয়েছি super কিওয়ার্ড এর মাধ্যমে।

এখন আমরা ride.py ফাইলটি ওপেন করে আরো একটি ক্লাস ক্রিয়েট করব **RideSharing** নামে এবং তার মধ্যে **add_rider(), add_driver()** এবং একটা **__str__** মেথড ইমপ্লিমেন্ট করব যেটা আমাদেরকে অব্জেক্ট এর স্ট্রিং রিপ্রেজেন্টেশন দিবে —

```
class RideSharing:
    def __init__(self, company_name) → None:
        self.company_name = company_name
        self.riders = []
        self.drivers = []
        self.rides = []

    def add_rider(self, rider):
        self.riders.append(rider)

    def add_driver(self, driver):
        self.drivers.append(driver)

    def __str__(self):
        return f"Company Name {self.company_name} with
        riders : {len(self.riders)} and Drivers :
        {len(self.drivers)}"
```

এখন আমরা এই প্রত্যেক্টা ফিচারকে একসাথে নিয়ে কাজ করব তারজন্য main.py নামে এক্টা ফাইল খুলে ফেলি তারপর আমরা যেই ক্লাসগুলো ইমপ্লিমেন্ট করছি সেগুলোকে ইম্পোর্ট করে ফেলি—

```
# in main.py
from ride import Ride, RideRequest, RideMatching, RideSharing
from users import Rider, Driver
from vehicle import Car, Bike
```

Ride Request function in Rider class :

এখন আমরা main.py ফাইলে কিছু অব্জেক্ট ক্রিয়েট করব

```
#in main.py

niye_jao = RideSharing("Niye Jao”)

rahim = Rider("Rahim Uddin", "rahim@gmail.com", 1234,
        "Mohakhali", 1200)

niye_jao.add_rider(rahim)

kolimuddin = Driver("Kolim Uddin", "kolim@gmail.com", 1256,
             “Gulshan”)

niye_jao.add_driver(kolimuddin)
```

এখানে **niye_jao** হচ্ছে আমাদের রাইড শেয়ারিং এপ যেখানে rahim হচ্ছে একজন রাইডার এবং **kolimuddin** হচ্ছে একজন ড্রাইভার rahim যখন এক্টা রাইড রিকুয়েস্ট দিবে **kolimuddin** অব্জেক্ট এর **reach_destination** মেথড কল হওয়ার মাধ্যমে রাইড শুরু হবে

আমরা পুর্বে ১১-২ এ **request_ride** মেথড ইমপ্লিমেন্ট করিনি এখন সেটাকে ইমপ্লিমেন্ট করে ফেলি-

```
# in users.py
from ride import RideRequest, RideMatching

def request_ride(self, ride_sharing, destination,
vehicle_type):
        ride_request = RideRequest(self, destination)
        ride_matching = RideMatching(ride_sharing.drivers)
        ride = ride_matching.find_driver(ride_request,
        vehicle_type)
        ride.rider = self
        self.current_ride = ride
        print("YAY!! We got a ride")
```

এখন Ridematching করার জন্য আমরা ride.py এর মধ্যে RideMatching ক্লাস এর find_driver মেথডটিকে এইভাবে আপডেট করে ফেলি-

```
# in ride.py
class RideMatching:
    ....
    ....

    def find_driver(self, ride_request, vehicle_type):
        if len(self.available_drivers) > 0:
            print("Looking for drivers.....")
            driver = self.available_drivers[0]

            if vehicle_type == 'car':
                vechicle = Car('car', "ABC456", 30)
            elif vehicle_type == 'bike':
                vechicle = Bike("bike", "1234BH", 50)
            ride = Ride(ride_request.rider.current_location,
                        ride_request.end_location, vechicle)
            driver.accept_ride(ride)
            return ride
```

এবং আমরা ride.py এর মধ্যেক Ride ক্লাস এর মধ্যে আরো একটি এট্রিবিউট এড করব vehicle type টাকে রাখার জন্য-

```
# in ride.py

class Ride:
    def __init__(self, start_location, end_location, vehicle)
    → None:
        self.start_location = start_location
        self.end_location = end_location
        self.driver = None
        self.rider = None
        self.start_time = None
```

```
        self.end_time = None
        self.estimated_fare =
        self.calculate_fare(vehicle.vehicle_type)

        self.vehicle = vehicle
```

এই মডিউল এই পর্যন্তই নেক্সট এ আমরা আরো কিছু ফাংশনালিটি ইমপ্লিমেন্ট করব

Calculate Fare :

আমরা মোটামুটি অনেকগুলো কাজ করে ফেলছি এখন আমরা আরো একটি ইম্পর্ট্যান্ট কাজ করব সেটা হচ্ছে ভাড়া **(fare)** ক্যালকুলেট করা তারজন্য আমরা ride.py এ Ride ক্লাসএর মধ্যে **calculate_fare()** নামে একটা মেথড ডিক্লেয়ার করি -

```
def calculate_fare(self, vehicle):
        print(vehicle)
        distance = 10
        fare_per_km = {
            'car' : 30,
            'bike' : 20,
            'cng' : 25
        }
        return distance*fare_per_km.get(vehicle)
```

এই মেথডে আমরা কিলোমিটার প্রতি ভেহিকল অনুযায়ি ভাড়া নিয়ে একটা ডিকশনারি ডিফাইন করেছি এখন আমরা main.py থেকে এটাকে এইভাবে কল করে দেখতে পারি-

```
...
...
rahim.request_ride(niye_jao, "Uttara", "car")
kolimuddin.reach_destination(rahim.current_ride)
rahim.show_current_ride()
```

এইখানে `reach_destination` কে সঠিকভাবে কাজ করানোর জন্য `users.py` ফাইলে `Driver` ক্লাস এর মেথডটাকে এইভাবে আপডেট করতে হবে-

```
class Driver(User):
    ...
    ...
    ...
    def accept_ride(self, ride):
        ride.start_ride()
        ride.set_driver(self) # driver er object

    def reach_destination(self, ride):
        ride.end_ride()
```

এতটুকু করলেই আমাদের রাইড শেয়ারিং প্রজেক্ট এর কাজ মোটামুটি শেষ। এখন প্রাক্টিস এর জন্য এই প্রজেক্ট এর মধ্যে তোমার নিজের আরো কিছু আইডিয়া ইমপ্লিমেন্ট করা এবং While লুপ ইউস করে একটা ইউজার ফ্রেন্ডলি রেপ্লিকা সিস্টেম তৈরি করতে পারো।

The End