PYTHON

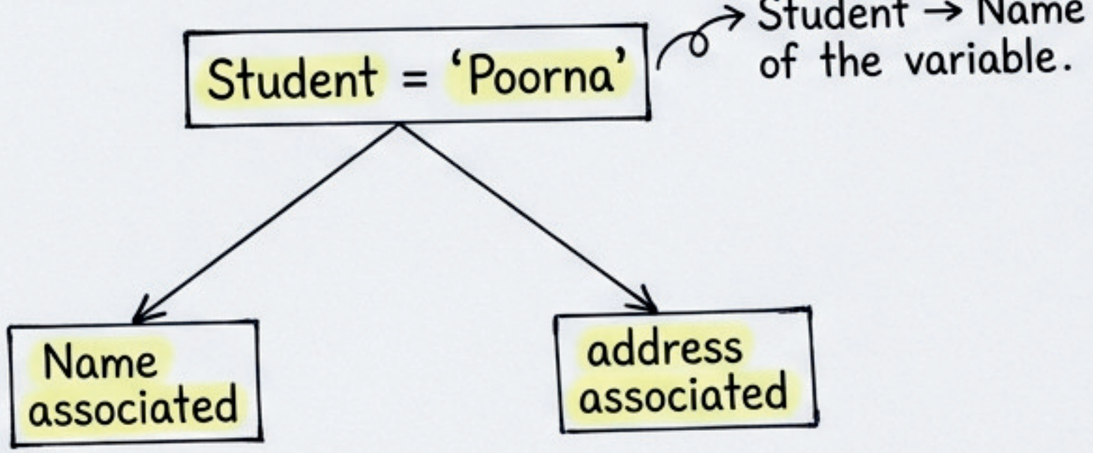
I want to speak with the computer.

Language = Python ⟶ Guido van Rossum - 1991

Syntax : import pandas as pd

Programming language application : Gaming, Banking, Machine learning.

Basic variables.



Data → collection of the facts.

Ex : My name is poorna.

How to store the data		
'Poorna'	→	String
56	→	Integer
True/False	→	Boolean

Data types in python (The Atoms)

1) Integers (int) : 1, 2, 3, 4, 5, 100, 1000, 999
2) Float : 6·19, 3·14, 9·32, 8·674
3) Boolean : True, False
4) String : 'Poorna', 'Chandru', 'Earth' (mention it inside double quote)
5) Complex number : a1 = 6+9j (Both Real and Imaginary part).

code checks

```
a1=100 → type(a1) → int
a1=3.14 → type(a1) → Float
a1=True → type(a1) → Bool
a1='Poorna' → type(a1) → Str
```

Operators in Python

Arithmetic Operators

```
a=10
b=20
```

a+b ⟶ 30 (10+20)
a-b ⟶ -10 (10-20)
b-a ⟶ 10 (20-10)
a*b ⟶ 200 (10*20)
a/b ⟶ 0.5 (10/20) ⟶ We should use forward slash

Relational Operators (>, <, ==, !=)

a=10, b=20
a>b ⟶ False
b>a ⟶ True (20>10)
a<b ⟶ True (10<20)
a==b ⟶ False (10 is not equal to 20)
a!=b ⟶ True

a=100, b=100 ⟶ a==b ⟶ True

Logical Operators (and, or, not)

AND Operator
a=True(1) , b=False(0)
a & b ⟶ False
a & a ⟶ True

We get a true value only when both operands are true.

OR Operator
a | b ⟶ True
b | b ⟶ False

It will give us the true result when either of the operands is true or both are true.

Tokens & Strings

Tokens: Smallest meaningful component in a program.

- Keywords →

False	None	True	and	as	class
def	if	elif	else	return	lambda	yield

Special reserved words.

- Identifiers → Names used for variables, functions, or objects.
  - No special character except _
  - Case sensitive
  - First letter cannot be a digit.
  - Poorna is his Identifier.
- Literals
- Operators

Strings: Sequence of characters enclosed within quotes.

↳ b1 = 'Hello World'
↳ b1 = "This is poorna"
↳ b1 = """This is a multiline string"""

My_string[-1] → 'a'

```
  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16
My_string = 'My_name_is_poorna'
-17 -16 -15 -14 -13 -12 -11 -10 -9 -8 -7 -6 -5 -4 -3 -2 -1
```

My_string[0] → 'M'

Slicing: [3:7] → 'name'

[11:17] → 'Poorna'

len(My_string) → 17

upper() → 'MY NAME IS POORNA'

split(',') → ['apple', 'mango']

Tuples in Python ( )

Ordered collection of elements enclosed within ().
Heterogeneous mixture of different elements.

Tuples are Immutable. Once you create the tuple, you can't change the value inside.

```
tup1 = (100, 'b', True, 'c', False)
```

Extraction

tup1[0] → 100
tup1[-1] → False
tup1[1:3] → 'b', True

Error check: tup1[2] = 'hello' → Error! Tuple object does not support item assignment.

Operations

Concatenating: tup1 + tup2 → (1, 2, 3, 4, 5, 6)
Repeating: tup1 * 3 → ('Poorna', 300, 'Poorna', 300...)
len(tup1) → 5
min(tup1) → 1
max(tup1) → 5

Lists in Python []

Ordered collection of elements enclosed within [].

Lists are mutable. We can add, subtract, multiply values inside the list.

Creation & Example: L1 = [1, 'Poorna', 3.14, True, 5+9j]
int, str, float, bool, complex

Modifying a List:

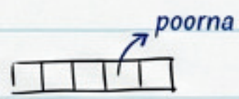
L1[0] = 100

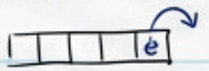
100

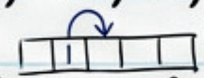
L1 = [~~1~~, 'a', 2, 'b'] ⟶ Result → [100, 'a', 2, 'b']

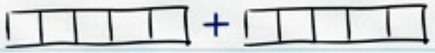
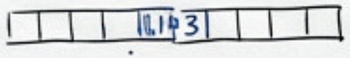
Key Operations:

- Appending: L1.append('poorna') → [..., 'poorna']
- Popping: L1.pop() → Removes last element
- Reversing: L1.reverse() → ['c', 3, 'b', 2, 'a', 1]
- Inserting: L1.insert(1, 'poorna')
- Sorting: L1.sort() → ['banana', 'cherry', 'guava'] (Alphabetical order).
- Concatenating: L1 + L2
- Repeating: L1 * 3





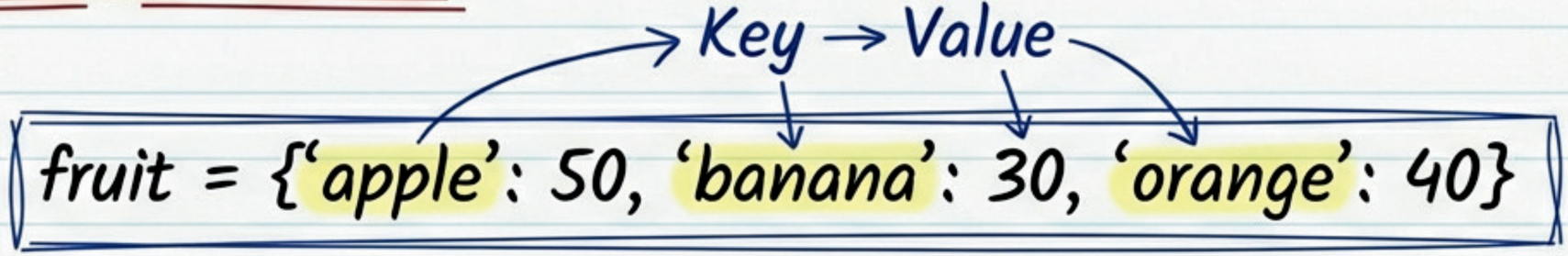


Dictionary in Python {}

Unordered collection of key-value pairs enclosed with {}.

Dictionary is Mutable.



Methods:
- fruit.keys() → ['Apple', 'orange', 'banana']
- fruit.values() → [50, 30, 40]
- fruit.items() → [('apple', 50), ('banana', 30)...]

Modifying Elements:
- Adding: fruit['Mango'] = 50
- Changing: fruit['Apple'] = 100
- Update: fruit1.update(fruit2)
- Popping: fruit.pop('orange')

Sets in Python {}

Unordered and unindexed collection enclosed with {}.

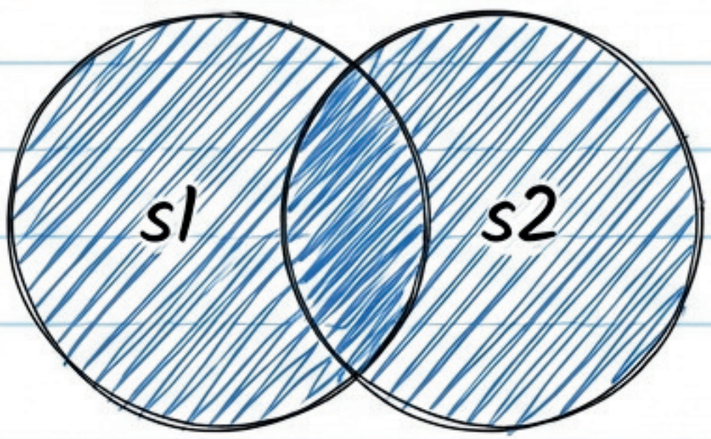
Rule: Duplicates are not allowed.

s1 = {1, 'Poorna', 'Poorna', 1} ⟶ Result -> {1, 'poorna'} (Duplicates removed)

Operations:

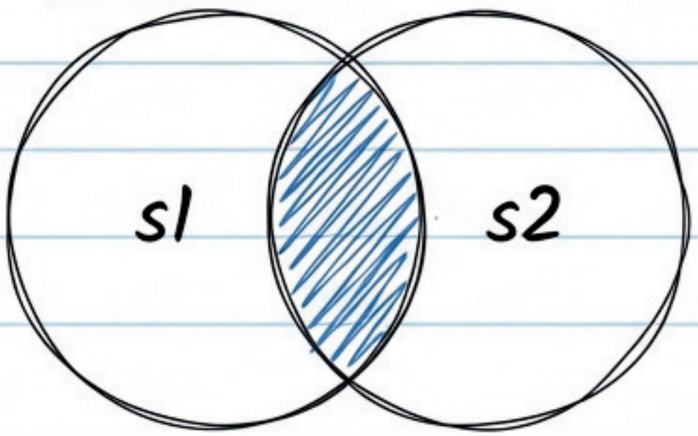
- s1.add('Hello') ⟶ Add one element.
- s1.update([10, 20, 30]) ⟶ Add multiple elements from list/set.
- s1.remove('b') ⟶ Removes element 'b', error if not present.

s1.union(s2)



Basically concatenating two sets.

s1.intersection(s2)



To find out the common elements

Decision Making Statements

If...Else Syntax:

```
if a > b: print('a is greater')
else: print('a is not greater')
```

Elif Syntax (Multiple conditions):

```
if a > b and a > c: 'a is greatest'
elif b > a and b > c: 'b is greatest'
else: 'c is greatest'
```

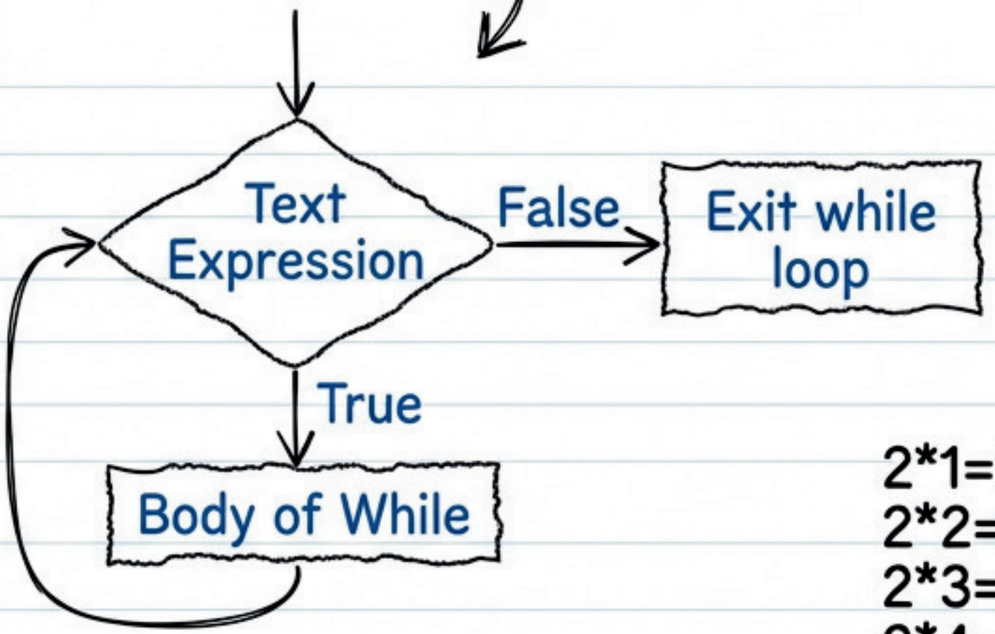
Conditionals with Data Structures

- ✓ Tuple: `if 2 in tup1: print('2 is present')`
- ✓ List: `if L1[1] == 2: L1[1] = L1[1] + 100` → `[1, 102, 3...]`
- ✓ Dictionary: `if d1['b'] == 2: d1['b'] = d1['b'] + 100`

Looping Statements

Used to repeat a task multiple times.

While Loop



* n=2, i=1

```
while i<=10:
    print(n, '*', i, '=', n*i);
    i=i+1
```

→

```
2*1=2
2*2=4
2*3=6
2*4=8
2*5=10
2*6=12
2*7=14
2*8=16
2*9=18
2*10=20
```

For Loop

Used to iterate over a sequence (tuple, list, dictionary).

Syntax:
```
for val in sequence:
    Body of for
```

* L1=['orange', 'blue'], L2=['book', 'chair']

```
for i in L1:
    for j in L2:
        print(i, j)
```

→

```
orange book
orange chair
blue book
blue chair
```

Basic Problems in Python

Caveat

1. Even or Odd

```
if num % 2 == 0:
    print('Even')
else:
    print('Odd')
```

→ Example: Input 5 → Odd
Input 8 → Even

Caveat

2. Factorial

```
factorial = 1
for i in range(1, num+1):
    factorial = factorial * i
```

Input 4 → 4 * 3 * 2 * 1 = 24

Caveat

3. Palindrome (e.g., 12321)

```
temp = n
rev = 0
while n > 0:
    dig = n % 10
    rev = rev * 10 + dig
    n = n // 10
if temp == rev: ...
```

Trace Table

n	dig	rev
121	1	1
12	2	12
1	1	121

Result: temp == rev → Palindrome

Caveat

4. Fibonacci (0 1 1 2 3 5 8)

```
a = 0; b = 1
for i in range(n):
    print(a)
    c = a + b
    a = b; b = c
```

Trace Table

a	b	c (a+b)
0	1	1
1	1	2
1	2	3

Functions & Lambda

Section 1: Functions

Block of code which performs a specific task.

Syntax: def function_name(): execute statements

```
def add10(x):
    return x+10
```

→ add10(10) → 20

Section 2: Lambda Functions

Anonymous function. Syntax: lambda arguments : expression

```
g = lambda x: x*x*x
```
→ print(g(7)) → 343

Section 3: High-Order Functions (Filter, Map, Reduce)

```
Li = [5, 7, 22, 97, 54, 10, 44]
```

Filter: → list(filter(lambda x: (x%2!=0), Li)) → Extracts odd numbers [5, 7, 97]

Map: → list(map(lambda x: x*2, Li)) → Doubles every number [10, 14, 44]

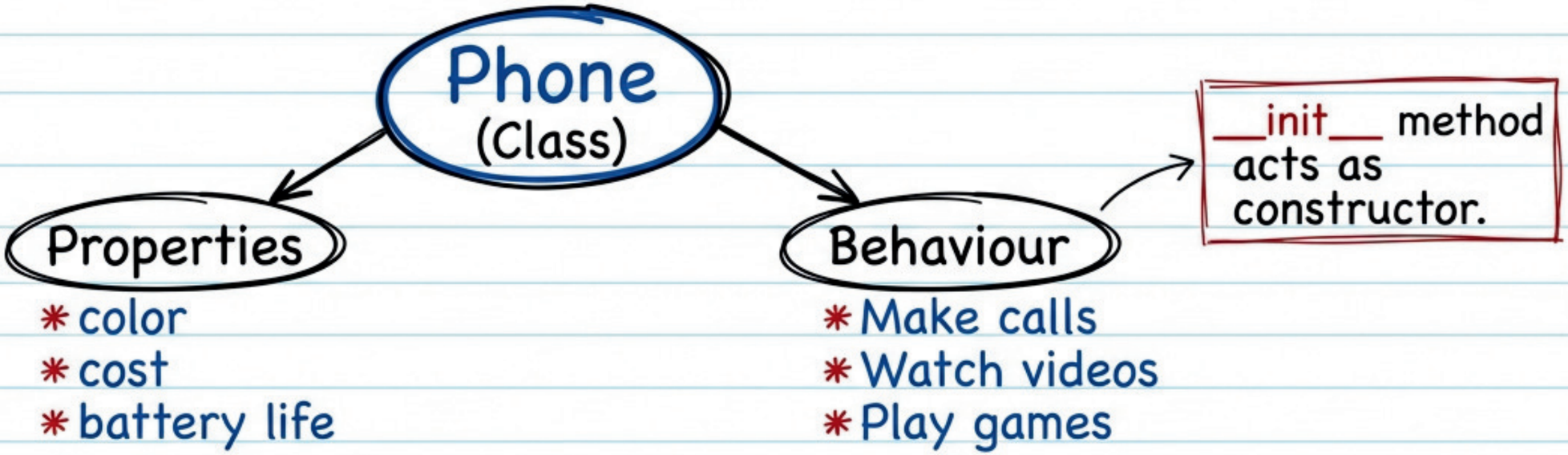
Reduce:

(from functools import reduce)

→ reduce((lambda x,y: x+y), Li) → Sums the list → 193

Object Oriented Programming (OOP)

- I am surrounded with objects: Mobile, laptop, bike, pen.
- Class is a template for real world entities.



Objects are specific instances of a class.
Class Phone → Objects: Apple, Samsung, Nokia

```
class phone:
    def make_call(self): print(`Making phone call')
p1 = phone()          <-- Instantiating object
p1.make_call()        <-- Invoking method
```

Inheritance in Python

One class can derive the properties of another class.

Parent Class Code

```
class vehicle:
    def __init__(self, mileage, cost):
        self.mileage = mileage
        self.cost = cost
    def show_details(self):
        print("I am a vehicle")
        print("Mileage of vehicle is", self.mileage)
        print("Cost of vehicle is", self.cost)
```

Child Class Code

```
class car(vehicle):
    def show_car(self): print('I am a car')
```

Instantiation Visual

```
c1 = car(200, 1200)
```

→ Instantiating object for child class

```
c1.show_details()
```

→ Invoking parent method

```
c1.show_car()
```

→ Invoking child method

Super Method (Important)

Over-riding init method:

```
def __init__(self, mileage, cost, tyres, hp):
    super().__init__(mileage, cost)
    self.tyres = tyres
    self.hp = hp
```

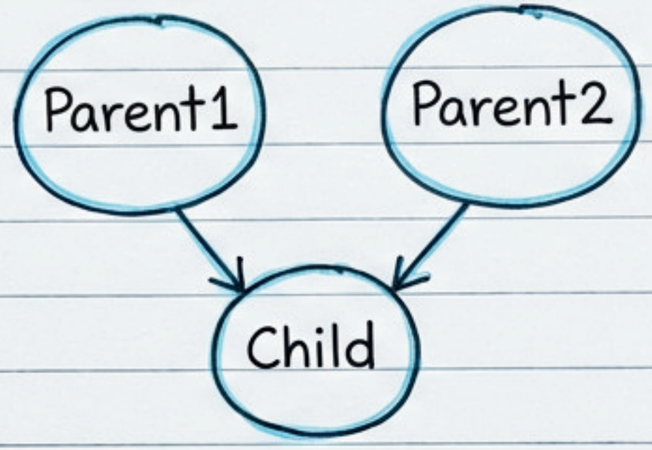
→ Invoking parent constructor

Types of Inheritance

Multiple Inheritance

Child inherits from MORE than 1 parent class.

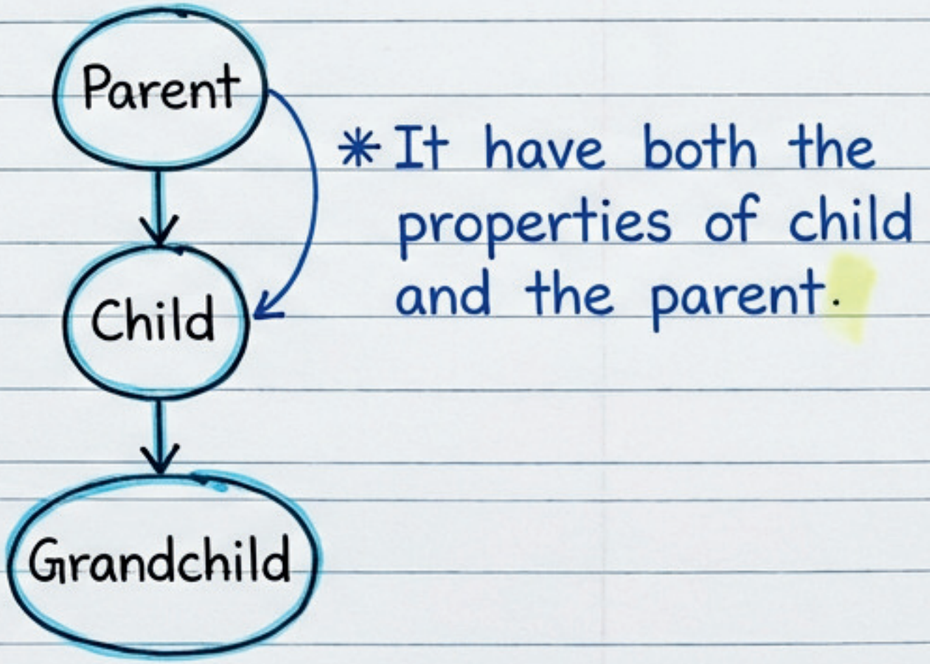
```
class derived(Parent1, Parent2):
```



Multi-level Inheritance

Parent, Child, and Grandchild relationship.

```
class parent()
class child(parent)
class grandchild(child)
```



Execution Example

```
g1 = grandchild()
g1.assign_name('Poorna')
g1.show_name() → Poorna
```

Arrays & Exception Handling

Arrays

Caveat: All the value of same type.

```
import array as arr
```

Type Codes	
Code	Description
'i'	signed integer
'I'	unsigned integer

```
vals = array('i', [5, 9, 8, 4, 2])
print(vals.buffer_info()) → (Address, size)

Looping: for i in range(len(vals)):
             print(vals[i])
```

Exception Handling

Types of Error:
1. Compile time (Syntax)
2. Runtime (Index Error)
3. Logical (4+4/2)

Keywords: try, except, else, finally

Code Block

```
try:
    num = int(input(...))
    print(math.factorial(num))
except ValueError:                    → e.g. negative input
    print('Cannot compute factorial of negative numbers')
```

User Defined Exception

```
class UserDefinedException(Exception): ...
    raise UserDefinedException('No empty string Allowed')
```

Throwing error manually

Throwing error manually