Python - Functional Programming

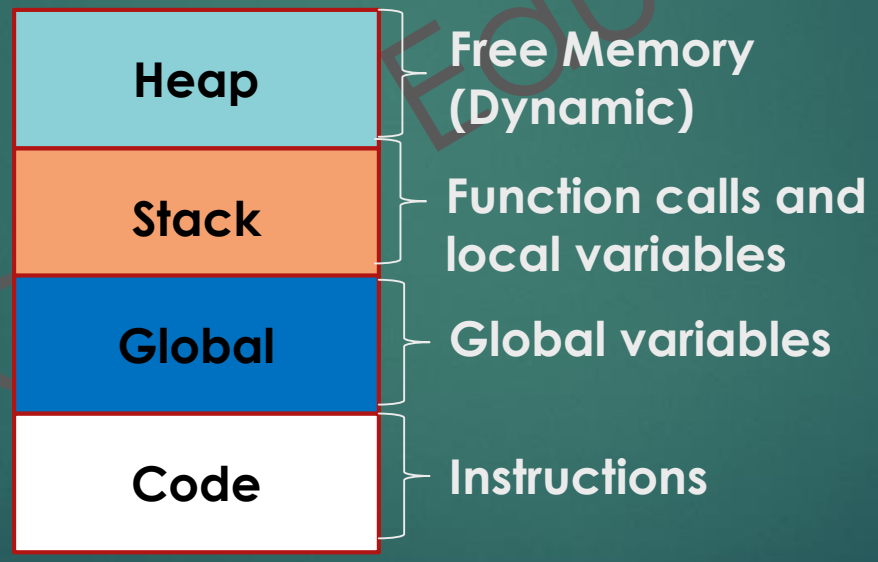
MEMORY MANAGEMENT (CPYTHON)

Contents

- id function
- References
- Garbage Collection Model
- Object Mutability
- How function arguments works internally?
- Variable Equality
- Optimizations

Memory

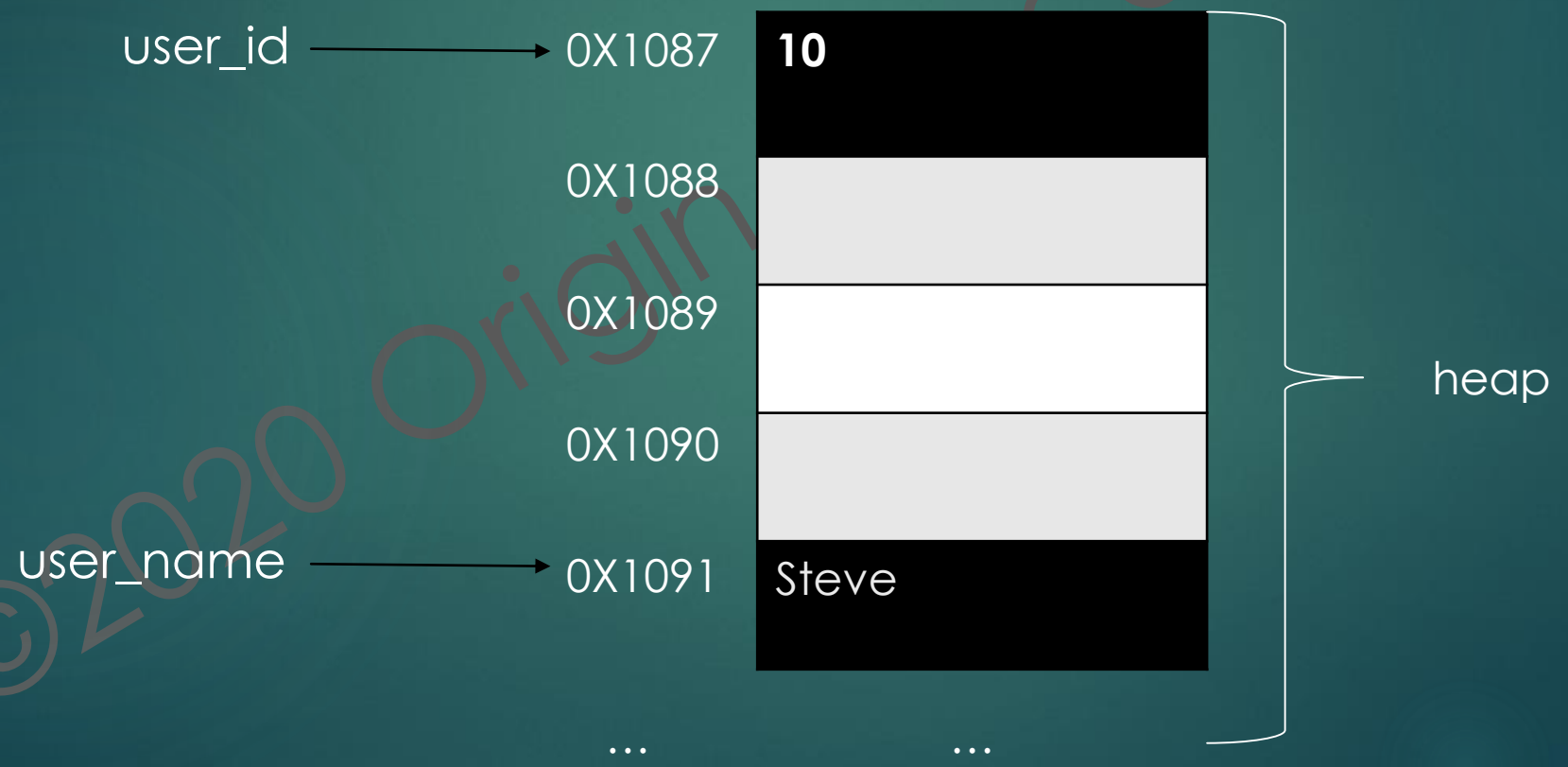
- Memory :
  - Stack for static memory allocation.
  - Heap for dynamic memory allocation.



Memory

user_id =10

user_name = ‘Steve’



id function

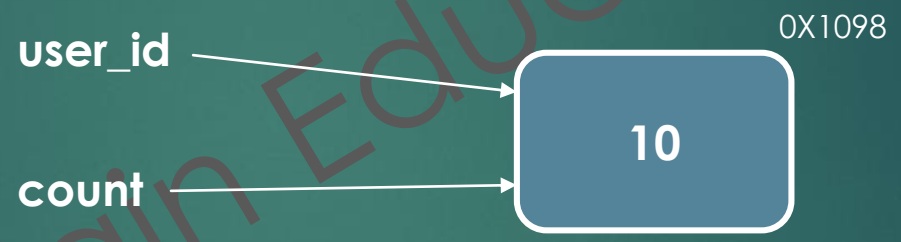
- In Python, we can find out the memory address referenced by a variable by using the id() function.
- Example user_id = 10 print(hex(id(user_id))) This will return a base-10 number.
- We can convert this base-10 number to hexadecimal, by using the hex() function.

Reference Counting

user_id = 10

count = 10

reference	count
0X1098	3



sys.getrefcount(user_id)

Note: By doing this one extra reference will be created.

- **Note:** Cpython-Reference_counting

Garbage Collector

- Can be controlled programmatically using the **gc** module.
- By default it is turned **on.**
- You may turn it off if you're sure your code does not create circular references – **beware!!**
- Runs periodically on its own (if turned **on**).
- You can call it manually, and even do your own cleanup.

Note: GC uses __del__() destructor on the object during cleanup, if the object is unused.

Ref: [Standar Garbage Collector interface]

Object Mutability

- An object whose internal state can be changed, is called **MUTABLE.**
- An object whose internal state cannot be changed, is called **IMMUTABLE.**

Immutable	Mutable
Numbers	Lists
Strings	Sets
Tuples	Dictionaries
Frozen Sets	User Defined Classes
User Defined Classes	

Object Mutability - Examples

t = (1, 2, 3) tuple is immutable

these are references to immutable object (int)

t = ([1, 2], [3, 4]) tuple is immutable

these are references to a mutable object (list)

Function Args and Mutability

Immutable objects are safe from unintended side-effects

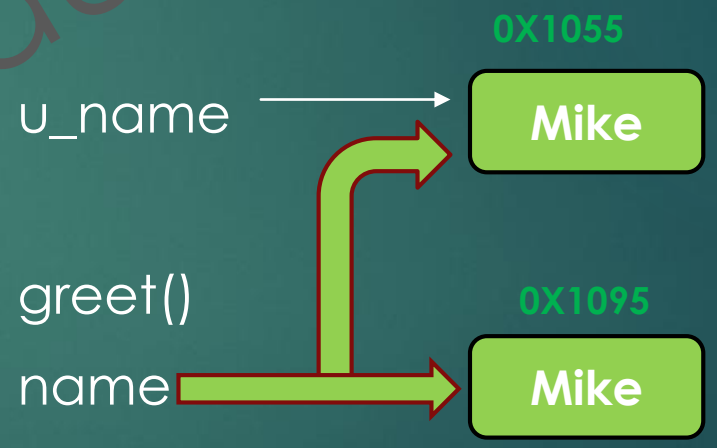
```
def greet(name):
    name = "Hello" +  name
    return name

u_name = "Mike"
greet(u_name)
```

0X1055

Behind the Scene:

module scope



Function Args and Mutability

Mutable objects are not safe from unintended side-effects

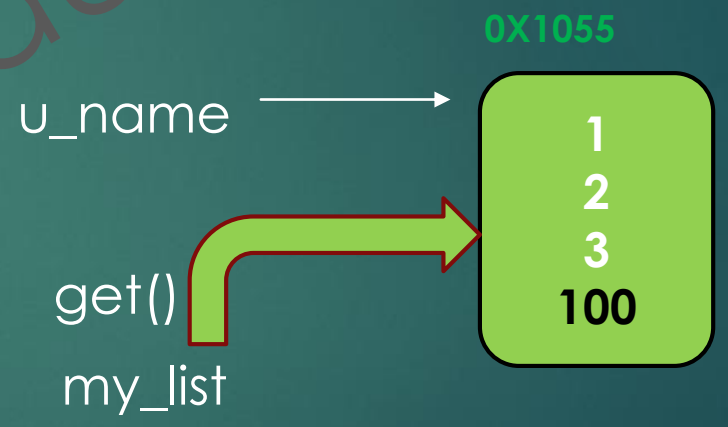
```
def get( lst ):
    lst.append(100)

my_list = [1,2,3]
get(my_list)
```

0X1055

Behind the Scene:

module scope



Variable Equality

- We can think of variable equality in two fundamental ways:

Memory Address

is

identity operator

item_1 is item_2

Object State(data)

==

equality operator

item_1 == item_2

Optimizations

- **Interning**:
  - reusing objects on-demand
- At startup, Python (CPython), **pre-loads** (caches) a global list of integers in the range [-5, 256].
- Any time an integer is referenced in that range, Python will use the cached version of that object this is termed as **Singleton Objects**.

When do the below:

customer_id = 10

Python just has to point to the existing reference for 10.

customer_id = 2581

Python does not use that global list and a new object is created every time

Optimization – Strings -1

- Some strings are also automatically interned.
- As the Python code is compiled, **identifiers** are interned.
  - variable names.
  - function names.
  - Class names,…
- Why to do :
- It's all about (speed and memory(not always)) optimization.
- Python, both internally, and in the code you write, deals with lots and lots of dictionary type lookups, on string keys, which means a lot of string equality testing.
- Let's say we want to see if two strings are equal:
- x = 'some_long_string' y = 'some_long_string'
- Using x == y, we need to compare the two strings **character by character**.

Optimization – Strings -2

- But if we know that 'some_long_string' has been interned, then x and y are the same string if they both point to the same memory address.
- This is **much faster** than the character by character comparison.

How to do it?

```
import sys
x = sys.intern( 'some_long_string')
y = sys.intern('some_long_string')
```

Now if we do x is y returns **True** [Faster way]

Note:

Identifiers:

- must start with _ or a letter
- can only contain _, letters and numbers

Behind the Scene:#1

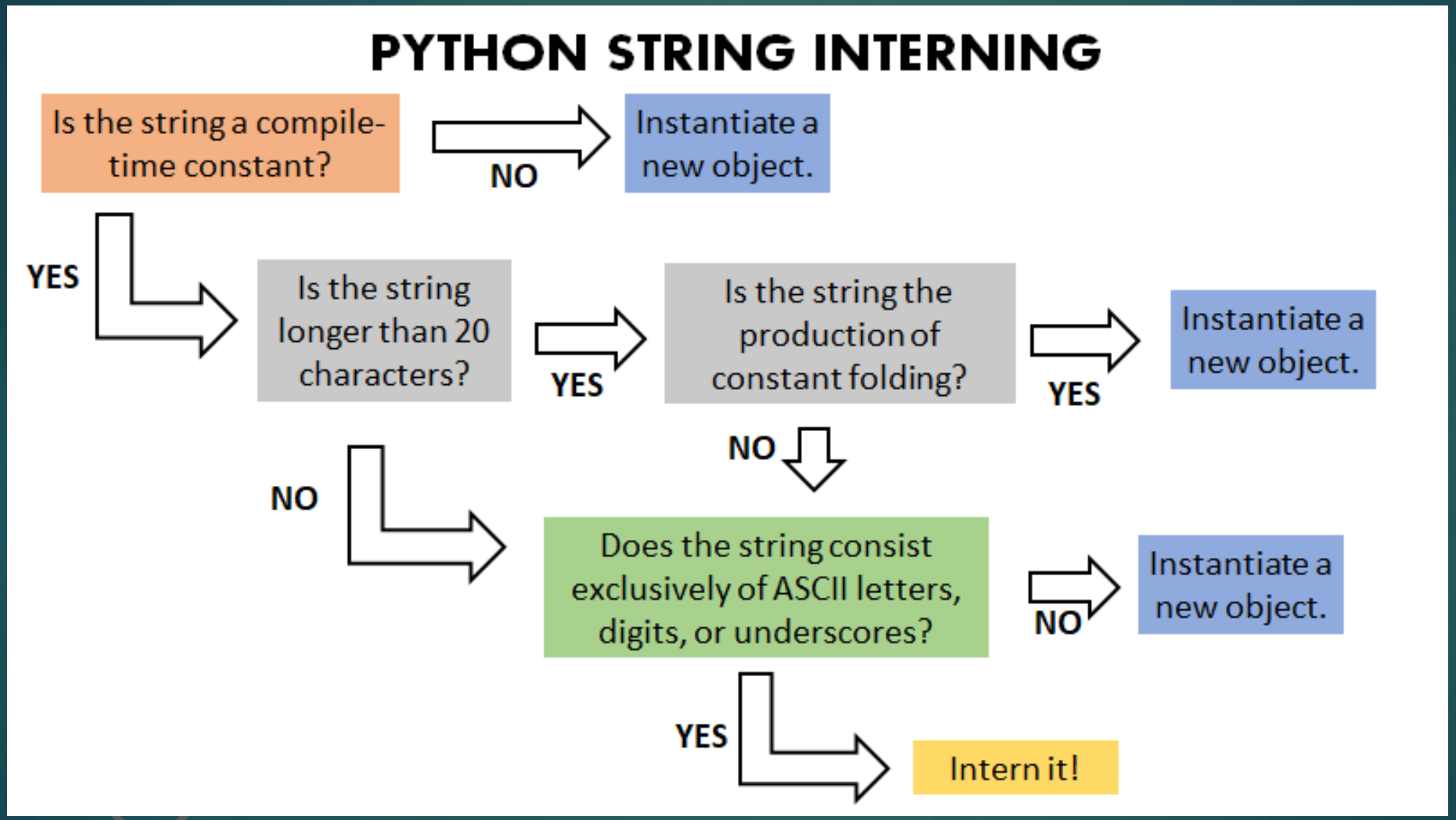


Image source : **Medium.com**

Behind the Scene:#2

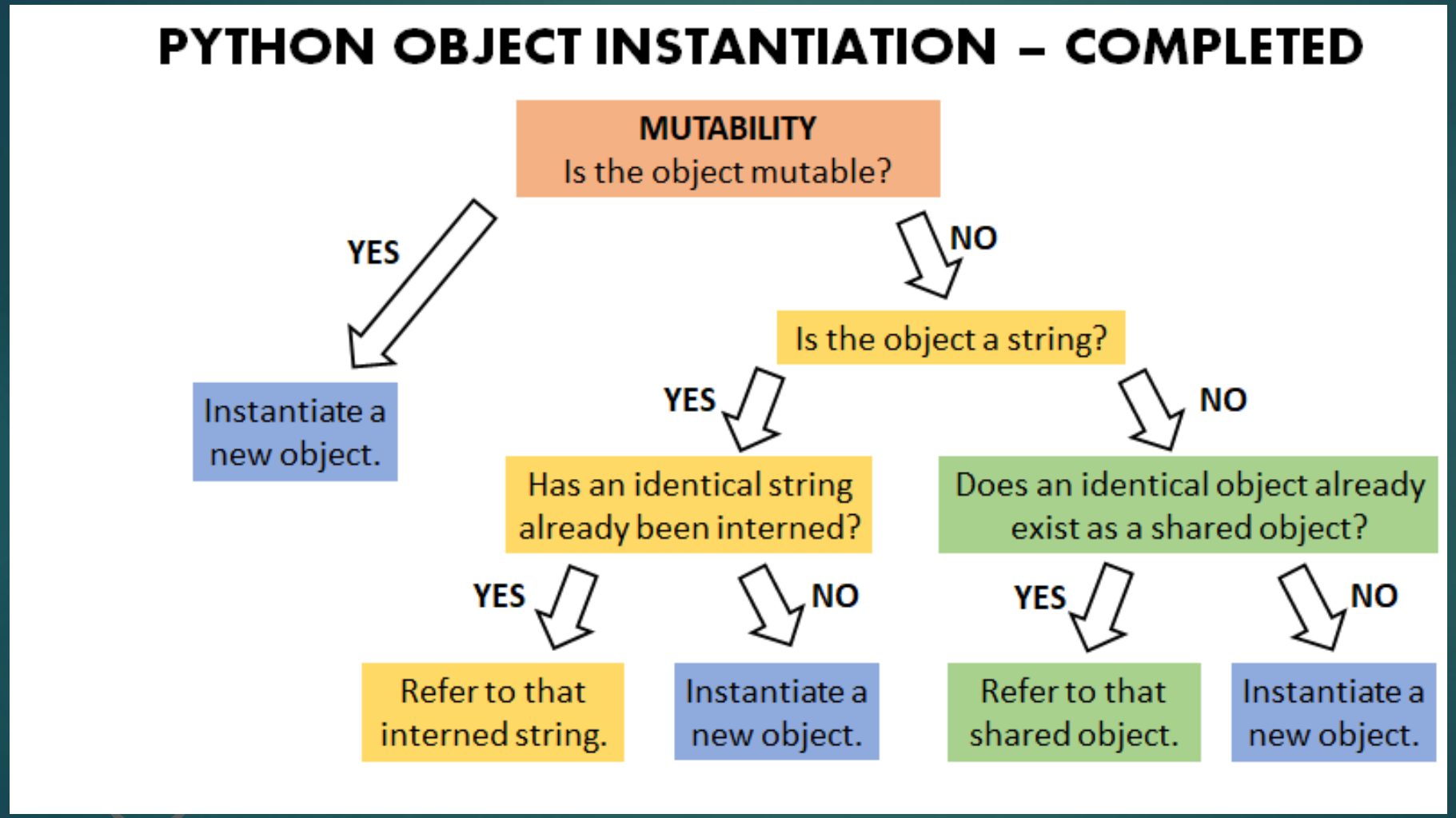


Image source : **Medium.com**

Info to Remember

- The **None** object
- The **None** object can be assigned to variables to indicate that they are not set (in the way we would expect them to be), i.e. an "empty" value (or null pointer).
- But the **None** object is a **real** object that is managed by the Python memory manager.
- Furthermore, the memory manager will always use a **shared reference** when assigning a variable to **None**.

send me your suggestions!

- Email: sundar.muthuraman.offical@gmail.com
- Contact me : +91 9962988838