

Python Series Questions-Answers

Watch Full Video On Youtube:

<https://youtube.com/playlist?list=PL1B8DuzqTNOJRjMrCrVd0a1WKyicperOb>

Connect with me:

Youtube: <https://www.youtube.com/c/nitmantalks>

Instagram: <https://www.instagram.com/nitinmangotra/>

LinkedIn: <https://www.linkedin.com/in/nitin-mangotra-9a075a149/>

Facebook: <https://www.facebook.com/NitManTalks/>

Twitter: <https://twitter.com/nitinmangotra07/>

Telegram: <https://t.me/nitmantalks/>

1. Difference Between List and Tuple.

LIST

1. Lists are **Mutable** datatype.
2. List is a container that contain different types of objects and is used to iterate objects.
3. To create an empty list, we use
`list1 = []` or `list2 = list()`
4. **Syntax Of List**
`list3 = ['a', 'b', 'c', 1,2,3]`
5. List iteration is slower
6. List is stored in two blocks of memory (One is fixed sized and the other is variable sized for storing data)
7. Creating a list is slower because two memory blocks need to be accessed.
8. Lists consume more memory.
9. An element in a list can be removed or replaced.
10. Lists are Growable.

Tuple

1. Tuples are **Immutable** datatype.
2. Tuple is also similar to list but contains immutable objects.
3. To create an empty tuple, we use
`tuple 1 = ()` or `tuple2 = tuple()`
4. **Syntax Of Tuple**
`tuple3 = ('a', 'b', 'c', 1, 2)`
5. Tuple processing is faster than List.
6. Tuple is stored in a single block of memory.
7. Creating a tuple is faster than creating a list because only one block of memory is needed.
8. Tuple consume less memory.
9. An element in a tuple cannot be removed or replaced.
10. Tuples are not Growable.

1.2. What is A List In Python?

- ❑ Lists are Mutable datatype in Python.
- ❑ List is a container that contain different types of objects and is used to iterate objects.
- ❑ In list, Duplicate and Heterogeneous objects are allowed.
- ❑ Insertion order is preserved in list.
- ❑ Lists are Growable.
- ❑ An element in a list can be removed or replaced.
- ❑ List is stored in two blocks of memory (One is fixed sized and the other is variable sized for storing data)
- ❑ Creating a list is slower because two memory blocks need to be accessed.
- ❑ Lists consume more memory as compared to tuple.
- ❑ To create an empty list, we use
`list1 = []` or `list2 = list()`
- ❑ Syntax Of List
`list3 = ['a', 'b', 'c', 1,2,3]`

1.3. What is A Tuple In Python?

- ❑ Tuples are **Immutable** datatype.
- ❑ Tuple is also a container similar to list but contains immutable objects.
- ❑ In Tuple, Duplicate and Heterogeneous objects are allowed.
- ❑ Insertion order is preserved in tuple.
- ❑ Tuples are not Growable.
- ❑ An element in a tuple cannot be removed or replaced.
- ❑ Tuple is stored in a single block of memory.
- ❑ Creating a tuple is faster than creating a list because only one block of memory is needed.
- ❑ Tuple consume less memory as compared to list.

- ❑ To create an empty tuple, we use
`tuple 1 = ()` or `tuple2 = tuple()`

- ❑ Syntax Of Tuple
`tuple3 = ('a', 'b', 'c', 1, 2)`

1.4 How To Define Empty List or Empty Tuple?

Empty LIST:

```
list1 = []
```

or

```
list2 = list()
```

Empty TUPLE:

```
tuple1 = ()
```

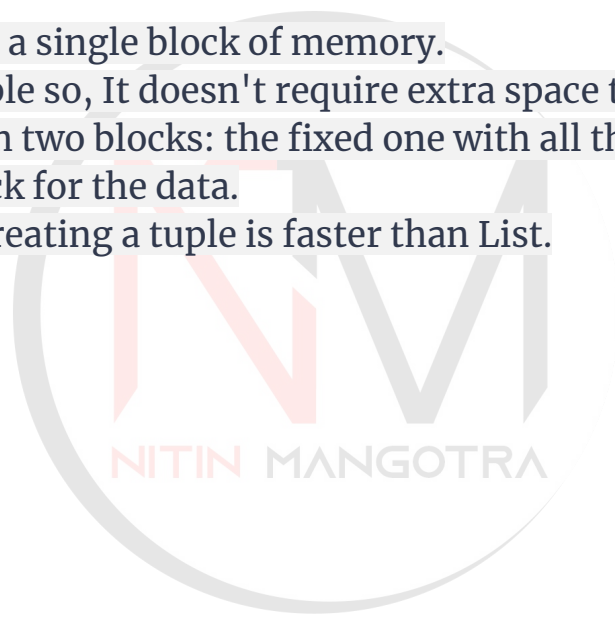
or

```
tuple2 = tuple()
```

NITIN MANGOTRA

1.5 Why Tuple Is Considered Faster As Compared To A List?

- ❑ Tuples are stored in a single block of memory.
- ❑ Tuples are immutable so, It doesn't require extra space to store new objects.
- ❑ Lists are allocated in two blocks: the fixed one with all the Python object information and a variable sized block for the data.
- ❑ This is the reason creating a tuple is faster than List.



1.6. Why Tuple Consume Less Memory? Give An Example To Prove It.

- ❑ Tuples are stored in a single block of memory.
- ❑ Tuples are immutable so, It doesn't require extra space to store new objects.
- ❑ Lists are allocated in two blocks: the fixed one with all the Python object information and a variable sized block for the data.
- ❑ Thus Tuple Consume less memory as compared to a list.

```
main.py x Console Shell
1 list1 = []
2 tuple1 = ()
3 print(type(list1))
4 print(list1.__sizeof__())
5 print(type(tuple1))
6 print(tuple1.__sizeof__())
```

```
<class 'list'>
40
<class 'tuple'>
24
>
```

1.6. Why Tuple Consume Less Memory? Give An Example To Prove It.

```

main.py x Console Shell
1 list1 = []
2 tuple1 = ()
3 print(type(list1))
4 print(list1.__sizeof__())
5 print(type(tuple1))
6 print(tuple1.__sizeof__())
7
8 print("<<<<----->>>>")
9
10 list2 = [1,2,3]
11 tuple2 = (1,2,3)
12 print(type(list2))
13 print(list2.__sizeof__())
14 print(type(tuple2))
15 print([tuple2.__sizeof__()])

```

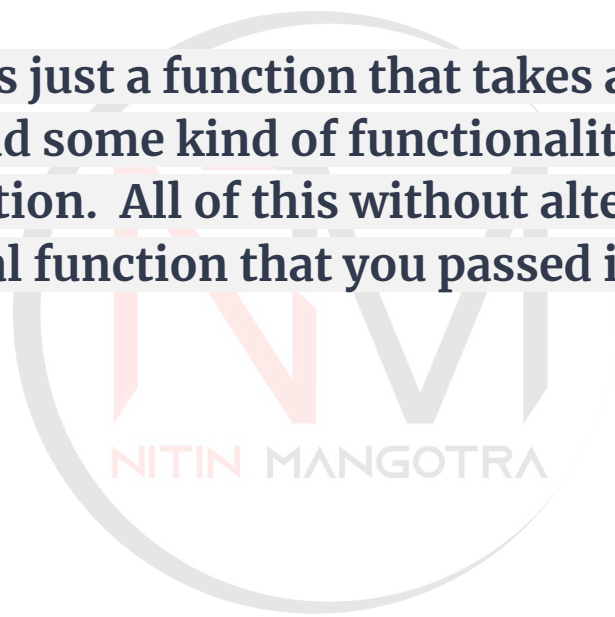
```

<class 'list'>
40
<class 'tuple'>
24
<<<<----->>>>
<class 'list'>
64
<class 'tuple'>
48

```


2. What is Decorator? Explain With Example.

- ❑ A Decorator is just a function that takes another function as an argument, add some kind of functionality and then returns another function. All of this without altering the source code of the original function that you passed in.



2.1. What is Decorator? Explain With Example.

A Decorator is just a function that takes another function as an argument, add some kind of functionality and then returns another function.

All of this without altering the source code of the original function that you passed in.

```
def decorator_func(func):
    def wrapper_func():
        print("wrapper_func Worked")
        return func()
    print("decorator_func worked")
    return wrapper_func
```

```
def show():
    print("Show Worked")
decorator_show = decorator_func(show)
decorator_show()
```

```
#Alternative
@decorator_func
def display():
    print('display
        worked')
display()
```

Output:
decorator_func worked
wrapper_func Worked
Show Worked
decorator_func worked
wrapper_func Worked
display worked

2.2. Have You Ever Use Decorator In Your Real Time Project? If Yes, Please Explain With An Example.

Yes! I have used a decorator for implementing Cron jobs. (Automated Jobs)

```
import time
from datetime import datetime

def log_datetime_decorator(func):
    '''Log the date and time of a function'''

    def wrapper_func():
        print(f'Function: {func.__name__}')

        print(f'Run on:
{datetime.today().strftime('%Y-%m-%d
%H:%M:%S')}')

        print(f'{"-----"}')
        func()
    return wrapper_func
```

```
@log_datetime_decorator
def daily_cron_job():
    #Do Anything Here
    time.sleep(9)
    print('Daily cron job has finished.')

@log_datetime_decorator
def weekly_cron_job():
    #Do Anything Here
    time.sleep(61.5)
    print('weekly cron job has finished.')

@log_datetime_decorator
def monthly_cron_job():
    #Do Anything Here
    time.sleep(2.1)
    print('monthly cron job has finished.')

daily_cron_job()
weekly_cron_job()
monthly_cron_job()
daily_cron_job()
```

Output:
Function: daily_cron_job
Run on: 2022-07-12 18:06:24

Daily cron job has finished.
Function: weekly_cron_job
Run on: 2022-07-12 18:06:33

weekly cron job has finished.
Function: monthly_cron_job
Run on: 2022-07-12 18:07:35

monthly cron job has finished.
Function: daily_cron_job
Run on: 2022-07-12 18:07:37

Daily cron job has finished.

2.2. Have You Ever Use Decorator In Your Real Time Project? If Yes, Please Explain With An Example.

Yes! I have used a decorator for implementing Cron jobs. (Automated Jobs)

main.py ×

```
1 import time
2 from datetime import datetime
3
4 def log_datetime_decorator(func):
5     '''Log the date and time of a function'''
6
7     def wrapper_func():
8         print(f"Function: {func.__name__}")
9         print(f"Run on: {datetime.today().strftime('%Y-%m-%d %H:%M:%S')}")
10        print(f'{"-----"}')
11        func()
12    return wrapper_func
```

```
15 @log_datetime_decorator
16 def daily_cron_job():
17     #Do Anything Here
18     time.sleep(9)
19     print('Daily cron job has finished.')
20
21 @log_datetime_decorator
22 def weekly_cron_job():
23     #Do Anything Here
24     time.sleep(61.5)
25     print('weekly cron job has finished.')
26
27 @log_datetime_decorator
28 def monthly_cron_job():
29     #Do Anything Here
30     time.sleep(2.1)
31     print('monthly cron job has finished.')
32
33 daily_cron_job()
34 weekly_cron_job()
35 monthly_cron_job()
36 daily_cron_job()
```

2.2. Have You Ever Use Decorator In Your Real Time Project? If Yes, Please Explain With An Example.

Yes! I have used a decorator for implementing Cron jobs. (Automated Jobs)

```
main.py x
1 import time
2 from datetime import datetime
3
4 def log_datetime_decorator(func):
5     '''Log the date and time of a function'''
6
7     def wrapper_func():
8         print(f"Function: {func.__name__}")
9         print(f"Run on:
10 {datetime.today().strftime('%Y-%m-%d %H:%M:%S')}")
11         print(f"{'-----'}")
12         func()
13     return wrapper_func
14
15 @log_datetime_decorator
16 def daily_cron_job():
17     #Do Anything Here
18     time.sleep(9)
19     print('Daily cron job has finished.')
20
21 @log_datetime_decorator
22 def weekly_cron_job():
23     #Do Anything Here
24     time.sleep(61.5)
25     print('weekly cron job has finished.')
26
27 @log_datetime_decorator
28 def monthly_cron_job():
29     #Do Anything Here
30     time.sleep(2.1)
31     print('monthly cron job has finished.')
32
33 daily_cron_job()
34 weekly_cron_job()
35 monthly_cron_job()
36 daily_cron_job()
```

Console Shell

```
Function: daily_cron_job
Run on: 2022-07-12 18:06:24
-----
Daily cron job has finished.
Function: weekly_cron_job
Run on: 2022-07-12 18:06:33
-----
weekly cron job has finished.
Function: monthly_cron_job
Run on: 2022-07-12 18:07:35
-----
monthly cron job has finished.
Function: daily_cron_job
Run on: 2022-07-12 18:07:37
-----
Daily cron job has finished.
```

2.3. Why We Use A Decorator?

- ❑ We'll use a decorator when we need to change the behavior of a function without modifying the function itself.
- ❑ You can also use one when you need to run the same code on multiple functions.
- ❑ A few good examples are when you want to add logging, authorization, test performance, perform caching, verify permissions, and so on.

NITIN MANGOTRA

2.4. How A Function Execute In Python?

A Function can only execute when we call the function using "()".
For example:



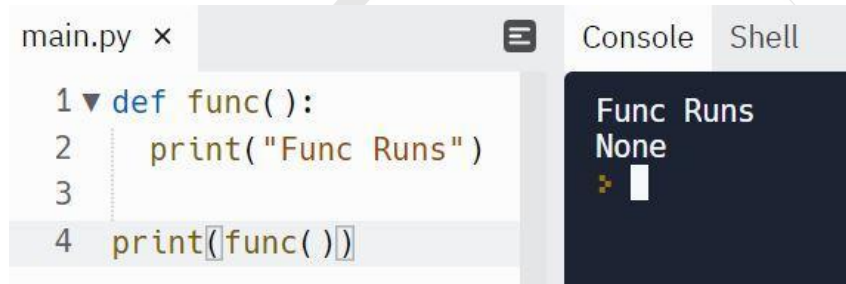
```
main.py x [Menu] Console Shell
1 ▼ def func():
2     print("Func Runs")
3
4 print(func)
```

The console output shows: `<function func at 0x7fa3f500d280>`

Here, At line 4, we didn't use "()" at the end of "func", thus we didn't get "Func Runs" in output.

2.4. How A Function Execute In Python?

Thus we need to use “()” at the end of the function for executing that function.
For example:



The screenshot shows a code editor with a file named 'main.py'. The code contains a function definition and a call to that function:

```
1 def func():
2     print("Func Runs")
3
4 print(func())
```

To the right of the code editor is a 'Console' tab. It displays the output of the program:

```
Func Runs
None
```

The output shows the string "Func Runs" followed by the value "None".

Here, We get the required output. But there is one “None” in the output, that is because we are not returning anything from that function.

2.4. How A Function Execute In Python?

Here we can return any value - Integer/String etc from that function.



```
main.py x [Menu] Console Shell
1 ▼ def func():
2     print("Func Runs")
3     return "Func Exit"
4
5 print(func())
```

The console output shows the function being called and returning a value:

```
Func Runs
Func Exit
>
```

This is how the complete function execute in Python.

2.5. Write Syntax For A Decorator.

The syntax for Creating and executing a decorator.

```
main.py ×
1 ▼ def decorator_func(func):
2 ▼     def wrapper_func():
3         # Do something before the function.
4         return func()
5         # Do something after the function.
6     return wrapper_func
```

```
@decorator_func
def my_func():
    # Do something after the function.
```

2.6. Walkthrough The Code.

```
def decorator_func(func):
    def wrapper_func():
        print("wrapper_func Worked")
        return func()
    print("decorator_func worked")
    return wrapper_func

def show():
    print("Show Worked")

decorator_show = decorator_func(show)
decorator_show()

#Alternative
@decorator_func
def display():
    print('display worked')

display()
```

1.) How A Function Execute In Python?

A Function can only execute when we call the function using "()".
For example:



The screenshot shows a code editor with a file named 'main.py'. The code contains a function definition and a call to the function. The console output shows the function object returned by the print statement.

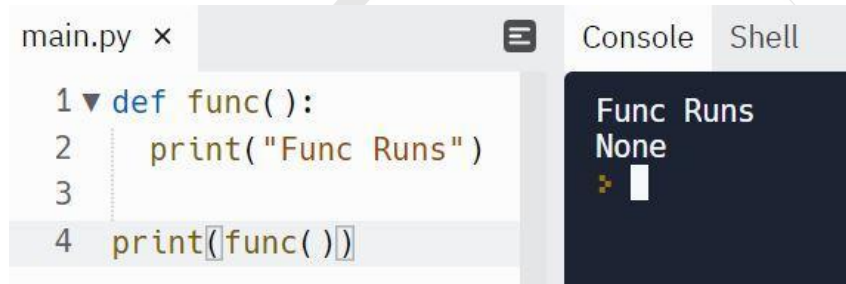
```
main.py x
1 def func():
2     print("Func Runs")
3
4 print(func)
```

Console: <function func at 0x7fa3f500d280>

Here, At line 4, we didn't use "()" at the end of "func", thus we didn't get "Func Runs" in output.

1.) How A Function Execute In Python?

Thus we need to use “()” at the end of the function for executing that function.
For example:



The screenshot shows a code editor with a file named 'main.py'. The code contains a function definition and a call to it:

```
1 def func():
2     print("Func Runs")
3
4 print(func())
```

To the right of the code editor is a 'Console' tab. It displays the output of the program:

```
Func Runs
None
```

The output shows the string "Func Runs" followed by "None" on a new line, which is the return value of the function.

Here, We get the required output. But there is one “None” in the output, that is because we are not returning anything from that function.

1.) How A Function Execute In Python?

Here we can return any value - Integer/String etc from that function.



```
main.py x [Menu] Console Shell
1 ▼ def func():
2     print("Func Runs")
3     return "Func Exit"
4
5 print(func())
```

The console output shows the function being called and returning a value:

```
Func Runs
Func Exit
>
```

This is how the complete function execute in Python.

2.) Everything In Python Is An Object

- ❑ As we all know, everything in python is an object. And so are functions.
- ❑ Thus, a function can be assigned to a variable.
- ❑ The function can be accessed from that variable.
- ❑ In similar way, like how we assign `a = 1`, we can assign `a = func` (Any function) also.

```
main.py x  Console  Shell
1 ▼ def func():
2     print('This Is Func')
3
4 # Assign the function to a variable without parenthesis.
5 # We don't want to execute the function.
6 func_variable = func
7
8 # Accessing the function from the variable I assigned it to.
9 func_variable ()
```

This Is Func

3.) A Function Can Be Nested Within Another Function.

main.py x

```
1 ▼ def outer_func():
2 ▼     def inner_func():
3         print('This Is Inner Func')
4
5     # Executing the inner func inside the outer func
6     inner_func()
7     print("This is End Of Outer Func")
8
9 outer_func()
```

Console Shell

```
This Is Inner Func
This is End Of Outer Func
❏
```


3.) A Function Can Be Nested Within Another Function.

Note: That the `inner_func` is not available outside the `outer_func`.

If I try to execute the `inner_func` outside of the `outer_func`, I receive a `NameError` exception.

main.py x

```
1 def outer_func():
2     def inner_func():
3         print('This Is Inner Func')
4
5     # Executing the inner func inside the outer func
6     inner_func()
7     print("This is End Of Outer Func")
8
9     inner_func()
10 # outer_func()
```

Console Shell

```
Traceback (most recent call last):
  File "main.py", line 9, in <module>
    inner_func()
NameError: name 'inner_func' is not defined
```

4.) A Function Can Return Another Function

main.py ×

```
1 ▼ def outer_func():
2 ▼     def inner_func():
3         print('This Is Inner Func')
4
5     return inner_func()
6     print("This is End Of Outer Func")
7
8 variable_outer_func = outer_func
9 variable_outer_func()
```

5.) A Function Can Be Passed To Another Function As An Argument

main.py ×

1▼ def outer_func(func):
2▼ def inner_func():
3 print('This Is Inner Func')
4 return func()
5
6 print("Before Return")
7 return inner_func
8
9▼ def argumental_func():
10 print("This Is Argumental Func")
11
12 variable_outer_func = outer_func(argumental_func)
13 print("Before Executing Variable Outer Func")
14 variable_outer_func()

Console

Shell

Before Return
Before Executing Variable Outer Func
This Is Inner Func
This Is Argumental Func
>

5.) A Function Can Be Passed To Another Function As An Argument

Alternative Way By Using "@"

```
main.py x  Console  Shell
1 ▼ def outer_func(func):
2 ▼   def inner_func():
3     print('This Is Inner Func')
4     return func()
5
6   print("Before Return")
7   return inner_func
8
9   @outer_func
10 ▼ def argumental_func():
11     print("This Is Argumental Func")
12
13   argumental_func()
```

```
Before Return
This Is Inner Func
This Is Argumental Func
>
```

3.1 What is Comprehension and How Many types of Comprehension are there in Python? Explain.

- ❑ Comprehensions in Python provide us with a short and concise way to construct new sequences (such as lists, set, dictionary etc.) using sequences which have been already defined.
- ❑ We can create new sequences using a given python sequence.
- ❑ This is called comprehension.

- ❑ Python supports the following 4 types of comprehensions:
 - List Comprehensions
 - Dictionary Comprehensions
 - Set Comprehensions
 - Generator Comprehensions

NOTE: *There is no TUPLE COMPREHENSION exists in Python

3.2 What is List Comprehension?

- ❑ List Comprehensions provide an elegant way to create new lists.
- ❑ List comprehensions must contain a for loop and can contain multiple for loops (nested list comprehensions).
- ❑ List comprehension may or may not contain an 'if condition'.

Syntax: [expression for item in iterable if conditional]

```
ls = [i*i for i in range(20) if i%2 != 0]
```

Here,

expression = $i*i$, required output (as element in list)

Item = i , iterable variable/ single member

Iterable = `range(20)`, any iterable thing - could be list/tuple/range etc

condition = $i\%2 \neq 0$, Any condition

3.2 What is List Comprehension?

Syntax: [expression for item in iterable if conditional]

Example 1: Create A List Of Odd Numbers From 0 to 20.

Common Way:

```
l = []  
for i in range(20):  
    if i%2 != 0:  
        l.append(i)  
print(l)
```

Output:

[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]

Using List Comprehension:

```
ls = [i for i in range(20) if i%2 != 0]  
print(ls)
```

Output:

[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]

3.2 What is List Comprehension?

Syntax: [expression for item in iterable if conditional]

Example 2: Multiply Elements of 2 List (use two for loops in list comprehension)

Common Way:

```
list1 = [2,3,4]
list2 = [12,13,14]

l = []
for i in list1:
    for j in list2:
        l.append(i*j)
print(l)
```

Output:

[24, 26, 28, 36, 39, 42, 48, 52, 56]

Using List Comprehension:

```
list1 = [2,3,4]
list2 = [12,13,14]

ls = [(i*j) for i in list1 for j in list2]
print(ls)
```

Output:

[24, 26, 28, 36, 39, 42, 48, 52, 56]

3.2 What is List Comprehension?

Syntax: [expression for item in iterable if conditional]

Example 3: Create a list containing list of pairs of elements from 2 different lists.
(use two for loops in list comprehension- Nested List Comprehension)

Common Way:

```
list1 = [2,3,4]
list2 = [12,13,14]
```

```
l = []
for i in list1:
    for j in list2:
        l.append([i,j])
print(l)
```

Output:

```
[[2, 12], [2, 13], [2, 14], [3, 12], [3, 13], [3, 14], [4, 12], [4, 13], [4, 14]]
```

Using List Comprehension:

```
list1 = [2,3,4]
list2 = [12,13,14]
```

```
ls = [[i,j] for i in list1 for j in list2]
print(ls)
```

Output:

```
[[2, 12], [2, 13], [2, 14], [3, 12], [3, 13], [3, 14], [4, 12], [4, 13], [4, 14]]
```

3.3 What is Dict Comprehension?

- ❑ we can also create a dictionary using dictionary comprehensions.

Syntax: {key:value for (key,value) in iterable if conditional}

Example 1: Create A Dict having square of Numbers From 1 to 10.

Common Way:

```
d = {}
for i in range(1,10):
    d[i] = i*i
print(d)
```

Output:

```
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

Using Dict Comprehension:

```
d1={i:i*i for i in range(1,10)}
print (d1)
```

Output:

```
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

3.3 What is Dict Comprehension?

Syntax: {key:value for (key,value) in iterable if conditional}

Example 2: Combine 2 lists to create a dictionary

Common Way:

```
dict1 = {}
list1 = [x for x in range(3)]
list2 = ['Mon', 'Tue', 'Wed']

for (key, value) in zip(list1, list2):
    dict1[key] = value

print(dict1)
```

Output:

```
{0: 'Mon', 1: 'Tue', 2: 'Wed'}
```

Using Dict Comprehension:

```
list1 = [x for x in range(3)]
list2 = ['Mon', 'Tue', 'Wed']

dict1 = {key:value for (key, value) in
zip(list1, list2)}

print(dict1)
```

Output:

```
{0: 'Mon', 1: 'Tue', 2: 'Wed'}
```

3.4 What is Set Comprehension?

- ❑ Set comprehensions are pretty similar to list comprehensions.
- ❑ The only difference between them is that set comprehensions use curly brackets { }.
- ❑ No Duplicate Values Allowed

Syntax: {expression for item in iterable if conditional}

Example 1: Create A List Of Odd Numbers From 0 to 20.

Common Way:

```
s1 = set()
for i in range(20):
    if i%2 != 0:
        s1.add(i)
print(s1)
```

Output:

{1, 3, 5, 7, 9, 11, 13, 15, 17, 19}

Using Set Comprehension:

```
s1 = {i for i in range(20) if i%2 != 0}
print(s1)
```

Output:

[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]

3.4 What is Set Comprehension?

Syntax: {expression for item in iterable if conditional}

Example 2: No Duplicate Values.

Common Way:

```
s1 = [1,2,3,4,2,3,2,3,4,1,5]
s2 = set()
for i in s1:
    if i%2 != 0:
        s2.add(i)
print(s2)
```

Output:
{1, 3, 5}

Using Set Comprehension:

```
s1 = [1,2,3,4,2,3,2,3,4,1,5]
s2 = {i for i in s1 if i%2 != 0}
print(s2)
```

Output:
[1, 3, 5]

3.5 What is Generator Comprehension?

- ❑ Generator Comprehensions are very similar to list comprehensions.
- ❑ One difference between them is that generator comprehensions use circular brackets "()" whereas list comprehensions use square brackets "["].
- ❑ The major difference between them is that generators don't allocate memory for the whole list.
- ❑ Instead, they generate each value one by one which is why they are memory efficient.

Syntax: (expression for item in iterable if conditional)

3.5 What is Generator Comprehension?

Syntax: (expression for item in iterable if conditional)

Example 1: Create A List Of Odd Numbers From 0 to 20.

Common Way:

```
def generator1(list1):
    for i in list1:
        if i%2 != 0:
            yield i

list1 = [1,2,3,4,5,6,7,8,9,10]
gen_values = generator1(list1)
print(gen_values)
print(next(gen_values))
print(next(gen_values))
```

Output:

```
<generator object generator1 at 0x7fec9e395e40>
1
3
```

Using Generator Comprehension:

```
gen_values = (i for i in range(20) if i%2 != 0)
print(gen_values)
print(next(gen_values))
print(next(gen_values))
```

Output:

```
<generator object <genexpr> at 0x7f64201c0e40>
1
3
```

3.5 What is Generator Comprehension?

Syntax: (expression for item in iterable if conditional)

Example 1: Create A List Of Odd Numbers From 0 to 20.

Common Way:

```
def generator1(list1):
    for i in list1:
        if i%2 != 0:
            yield i

list1 = [1,2,3,4,5,6,7,8,9,10]
gen_values = generator1(list1)
print(gen_values)
for i in gen_values:
    print(i, end = " ")
```

Output:

<generator object generator1 at 0x7fec9e395e40>
1 3 5 7 9

Using Generator Comprehension:

```
gen_values = (i for i in range(20) if i%2 != 0)
print(gen_values)
for i in gen_values:
    print(i, end = " ")
```

Output:

<generator object <genexpr> at 0x7f64201c0e40>
1 3 5 7 9

3.6 Compare Syntaxes Of All Comprehensions In Python.

List Comprehension

Syntax:

```
[expression for item in iterable if conditional]
```

Dict Comprehension

Syntax :

```
{key:value for (key,value) in iterable if conditional}
```

Set Comprehension

Syntax:

```
{expression for item in iterable if conditional}
```

Generator Comprehension

Syntax :

```
(expression for item in iterable if conditional)
```

3.7 Difference Between List and Dict Comprehension

List Comprehension

- We can also create a list using list comprehensions.
- List comprehensions use Square brackets [].

Syntax:

[expression for item in iterable if conditional]

Example Using List Comprehension:

```
ls = [i for i in range(10) if i%2 !=0]  
print(ls)
```

Output:

[1, 3, 5, 7, 9]

Dict Comprehension

- We can create a dictionary using dictionary comprehensions.
- Dict comprehensions use curly brackets { }.

Syntax :

{key:value for (key,value) in iterable if conditional}

Example Using Dict Comprehension:

```
d1 = {i:i*i for i in range(10) if i%2!=0}  
print(d1)
```

Output:

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

3.8 Difference Between List and Set Comprehension

List Comprehension

- We can also create a list using list comprehensions.
- List comprehensions use Square brackets [].
- Duplicate Values are allowed.

Syntax:

```
[expression for item in iterable if conditional]
```

Example Using List Comprehension:

```
ls = [i for i in range(10) if i%2 !=0]  
print(ls)
```

Output:

```
[1, 3, 5, 7, 9]
```

Set Comprehension

- We can create a set using set comprehensions.
- Set comprehensions use curly brackets { }.
- No Duplicate Values Allowed

Syntax :

```
{expression for item in iterable if conditional}
```

Example Using Dict Comprehension:

```
ls = {i for i in range(10) if i%2 !=0}  
print(ls)
```

Output:

```
{1, 3, 5, 7, 9}
```

3.9 Difference Between List and Dict Comprehension

List Comprehension

- We can also create a list using list comprehensions.
- List comprehensions use Square brackets [].

Syntax:

[expression for item in iterable if conditional]

Example Using List Comprehension:

```
ls = [i for i in range(10) if i%2 != 0]
print(ls)
```

Output:

[1, 3, 5, 7, 9]

Generator Comprehension

- Generator Comprehensions are very similar to list comprehensions.
- The major difference between them is that generators don't allocate memory for the whole list. Instead, they generate each value one by one which is why they are memory efficient.
- Generator comprehensions use circular brackets ()

Syntax :

{key:value for (key,value) in iterable if conditional}

Example Using Dict Comprehension:

```
gen_values = (i for i in range(20) if i%2 != 0)
print(gen_values)
for i in gen_values:
    print(i, end = " ")
```

Output:

<generator object <genexpr> at 0x7f64201c0e40>
1 3 5 7 9

3.10 Explain Tuple Comprehension In Python.

***There is no TUPLE COMPREHENSION exists in Python**

- ❑ Python supports the following 4 types of comprehensions only.:
- List Comprehensions
- Dictionary Comprehensions
- Set Comprehensions
- Generator Comprehensions

NITIN MANGOTRA

3.11 Why Are We Using Comprehensions In Python? Explain It With An Example.

- ❑ Comprehension is generally more compact and faster than normal functions and loops for creating list.
- ❑ In addition to standard list creation, list comprehensions can also be used for mapping and filtering. You don't have to use a different approach for each scenario.
- ❑ However, we should avoid writing very long list comprehensions in one line to ensure that code is user-friendly.

NITIN MANGOTRA

3.11 Why Are We Using Comprehensions In Python? Explain It With An Example.

Example 1: Which is taking more time -

Common Way:

```
import datetime
t1 = datetime.datetime.now()

for _list = []
for i in range(50000):
    for _list.append(i*i)

t2 = datetime.datetime.now()
print(t2-t1)
```

Output:

0:00:00.080154

Using List Comprehension:

```
import datetime
t1 = datetime.datetime.now()

for _list = []
comp_list = [i*i for i in range(50000)]

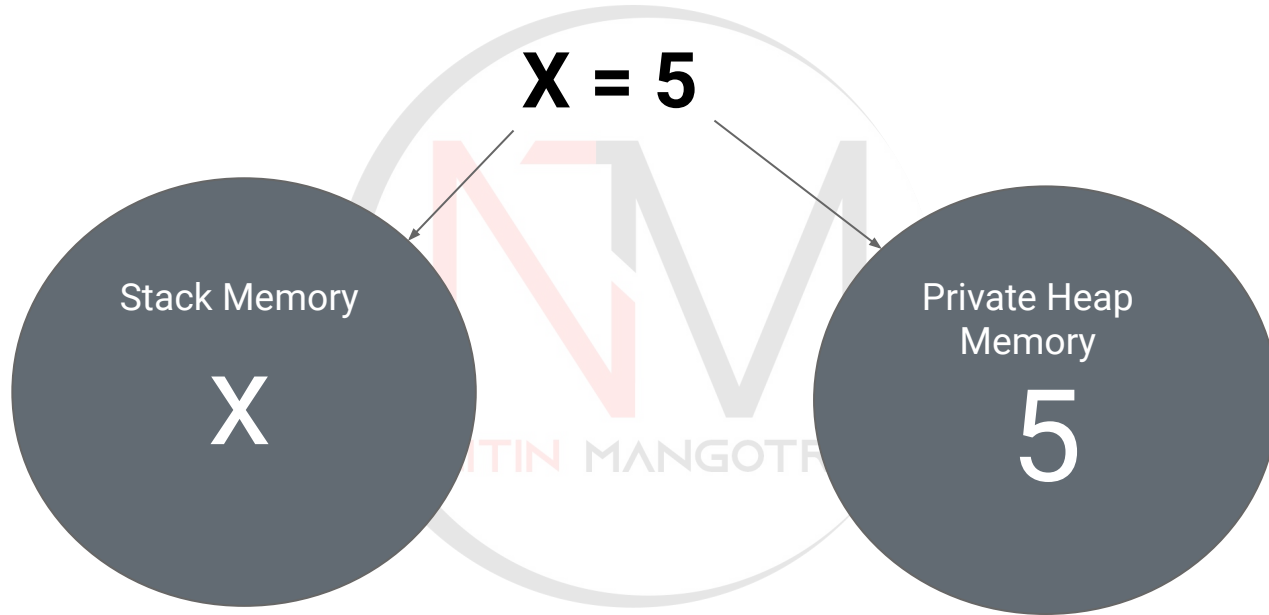
t2 = datetime.datetime.now()
print(t2-t1)
```

Output:

0:00:00.013842

As you can see in this example, List comprehension is taking lesser time as compared to normal way.

4. How Memory Managed In Python?



4.1 How Memory Managed In Python?

- ❑ Memory management in Python involves a **private heap** containing all Python objects and data structures. Interpreter takes care of Python heap and that the programmer has no access to it.
- ❑ The allocation of heap space for Python objects is done by **Python memory manager**. The core API of Python provides some tools for the programmer to code reliable and more robust program.
- ❑ Python also has a build-in garbage collector which recycles all the unused memory. When an object is no longer referenced by the program, the heap space it occupies can be freed. The garbage collector determines objects which are no longer referenced by the program frees the occupied memory and make it available to the heap space.
- ❑ The gc module defines functions to enable /disable garbage collector:
 - **gc.enable()** - Enables automatic garbage collection.
 - **gc.disable()** - Disables automatic garbage collection.

4.2 What is Garbage Collector & Reference Counting?

GARBAGE COLLECTOR

- ❑ Python deletes unwanted objects (built-in types or class instances) automatically to free the memory space.
- ❑ The process by which Python periodically frees and reclaims blocks of memory that no longer are in use is called **Garbage Collection**.
- ❑ Python garbage collector runs during program execution and is triggered when an object's reference count reaches zero.
- ❑ An object's reference count changes as the number of references referring that object changes
- ❑ The gc module (Garbage Collector Module) defines functions to enable /disable garbage collector:

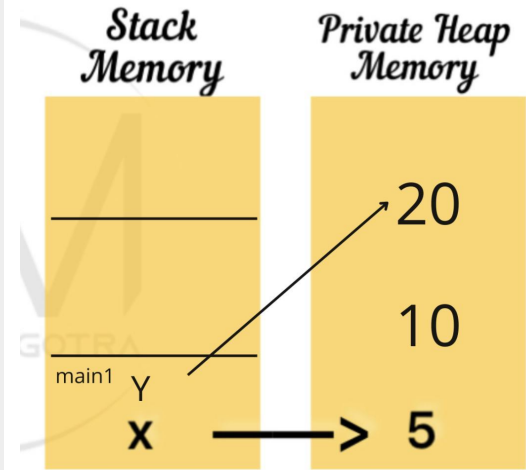
`gc.enable()` - Enables automatic garbage collection.

`gc.disable()` - Disables automatic garbage collection.

4.2 What is Garbage Collector & Reference Counting?

REFERENCE COUNTING

- ❑ Reference counting is one of the memory management technique in which the objects are deallocated when there is no reference to them in a program.
- ❑ The references are always counted and stored in memory.
- ❑ When the reference count of an object reaches 0, reference counting garbage collection algorithm cleans up the object immediately.
- ❑ In examples, 20 and 5 objects are being referred by X and Y. So there reference count would be 1 in both cases.
- ❑ Whereas in case of 10, there is no reference in stack memory, thus its reference counting is 0 (Zero).



4.3 What Is Duck Typing. And Why Python Is Called As Dynamic Typed Programming Language?

NOTE: The "**Duck typing**" name comes from the phrase, "If it walks like a duck and it quacks like a duck, then it must be a duck."

- Python don't have any problem even if we don't declare the type of variable.
- It states the kind of variable in the runtime of the program.
- Python also take cares of the memory management which is crucial in programming. So, Python is a dynamically typed language.

Variable a is assigned to a string

```
a = "NitMan Talks"  
print(type(a))
```

Output: <class 'str'>

Variable a is assigned to an integer

```
a = 7  
print(type(a))
```

Output: <class 'int'>

4.3 What Is Duck Typing. And Why Python Is Called As Dynamic Typed Programming Language?

```
x = 5
print(type(x)) # <class 'int'>
print(id(x)) # 140628347103296
```

```
y = x
print(type(y)) # <class 'int'>
print(id(y)) # 140628347103296
```

```
if (id(x)==id(y)):
    print("It's same")
# It's same
```

```
y = x
print(type(y)) # <class 'int'>
print(id(y)) # 140628347103296
```

```
x = x + 1
print(type(x)) # <class 'int'>
print(id(x)) # 140628347103328
```

```
if (id(x)!=id(y)):
    print("It's Different")
# It's Different
```

```
z = 5
print(type(z)) # <class 'int'>
print(id(z)) # 140628347103296
```

```
if (id(y)==id(z)):
    print("It's same")
else:
    print("It's Different")
# It's same
```

4.4 Which Function Is used To Enable/Disable Garbage Collector

The gc module (Garbage Collector Module) defines functions to enable /disable garbage collector:

`gc.enable()` - Enables automatic garbage collection.

`gc.disable()` - Disables automatic garbage collection.

NITIN MANGOTRA

4.5 DIFF BTW Stack & Heap Memory In Python.

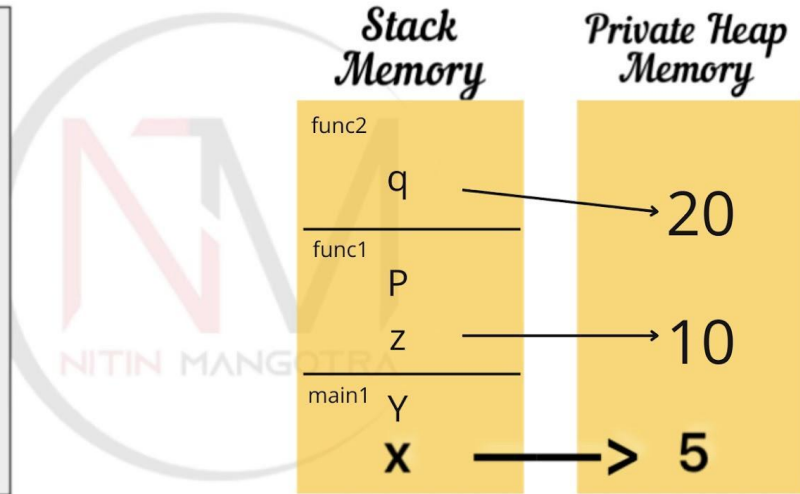
Stack Memory: The Methods/Method calls and the references are stored in stack memory.

Heap Memory: All The Values Objects Are stored in a private heap memory.

```
def func2(z):
    q = z + 10
    return q
```

```
def func1(x):
    z = x + 5
    p = func2(z)
    return p
```

```
#main
x = 5
y = func1(x)
print(y)
# 20
```



5.1 What Do You Mean By An Iterator In Python?

Explain It With An Example

- ❑ An iterator is an object which contains a countable number of values and it is used to iterate over iterable objects like list, tuples, sets, etc.
- ❑ Iterators are used mostly to iterate or convert other objects to an iterator using `iter()` function.
- ❑ Iterator uses `iter()` and `next()` functions.
- ❑ Lists, tuples, dictionaries, and sets are all iterable objects. They are iterable containers which you can get an iterator from.
- ❑ Technically, in Python, an iterator is an object which implements the iterator protocol, which consist of the methods `__iter__()` and `__next__()`.
- ❑ Every iterator is not a generator.

Example:

```
iter_list = iter(['A', 'B', 'C'])  
print(next(iter_list))  
print(next(iter_list))  
print(next(iter_list))
```

Output:

A
B
C

5.2 Why We Don't Call A List, Tupe, Dict Or Set An Iterator?

- ❑ Lists, tuples, dictionary, and sets are all iterable objects, and not Iterator
- ❑ They are iterable containers which you can get an iterator from.
- ❑ We can convert the Iterable objects into an iterator using “iter()”

main.py x

Console

Shell

```

1 list1 = [1,2,'NitMan', True]
2 # l1_iter = iter(list1)
3
4 print(next(list1))
5 print(next(list1))
6 print(next(list1))
7 print(next(list1))

```

```

Traceback (most recent call last):
  File "main.py", line 4, in <module>
    print(next(list1))
TypeError: 'list' object is not an iterator
>

```

5.3 Can We Use next() with List To Access Its Element?

- ❑ List is an iterable objects.
- ❑ Only Iterator can use next() function to access its elements and not iterable objects.
- ❑ Lists are iterable containers which you can get an iterator from.
- ❑ We can convert the list into an iterator using “iter()”.
- ❑ We will get below mentioned error if we use next() with list to access its elements.

main.py ×

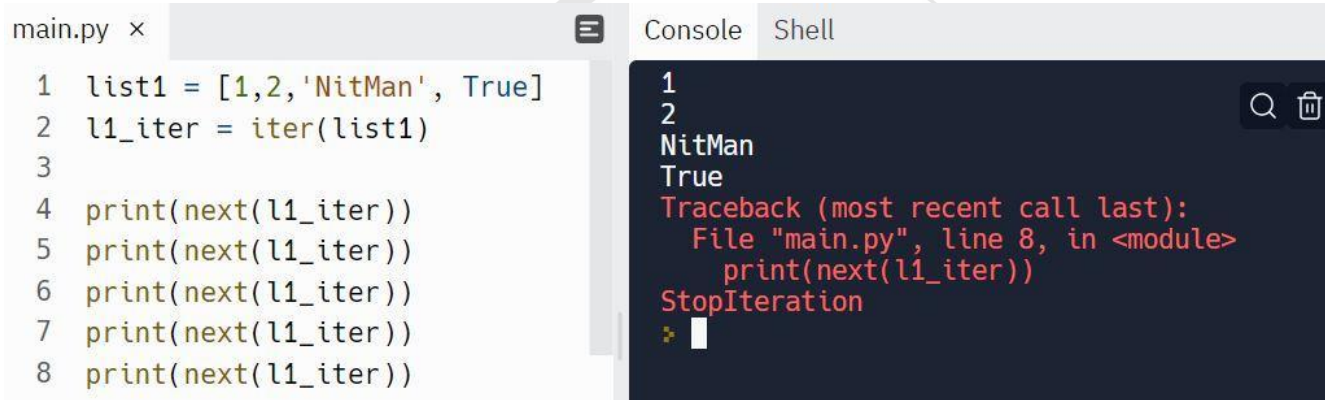
```
1 list1 = [1,2,'NitMan', True]
2 # l1_iter = iter(list1)
3
4 print(next(list1))
5 print(next(list1))
6 print(next(list1))
7 print(next(list1))
```

Console Shell

```
Traceback (most recent call last):
  File "main.py", line 4, in <module>
    print(next(list1))
TypeError: 'list' object is not an iterator
> |
```

5.4 What Happen When Will Try To Access An Item Which Is Not In The Range Of An Iterator.

We will get “STOP ITERATION” Error as mentioned below.



The screenshot shows a Python IDE with a file named 'main.py' and a console window. The code in 'main.py' creates a list 'list1' with elements [1, 2, 'NitMan', True], creates an iterator 'l1_iter' from it, and then prints the next element five times. The console output shows the first two elements, 'NitMan', and 'True', followed by a 'StopIteration' error. The error message indicates that the iteration has ended because there are no more elements to process.

```
main.py x [Console Shell]
1 list1 = [1,2,'NitMan', True]
2 l1_iter = iter(list1)
3
4 print(next(l1_iter))
5 print(next(l1_iter))
6 print(next(l1_iter))
7 print(next(l1_iter))
8 print(next(l1_iter))

1
2
NitMan
True
Traceback (most recent call last):
  File "main.py", line 8, in <module>
    print(next(l1_iter))
StopIteration
❖
```

6.1 Why do we use '___init___.py' file In Python?

___init___.py file

- ☐ The ___init___.py file lets the Python interpreter know that a directory contains code for a Python module.
- ☐ It can be blank.
- ☐ Without one, you cannot import modules from another folder into your project.
- ☐ The role of the ___init___.py file is similar to the ___init___ function in a Python class.
- ☐ The file essentially the constructor of your package or directory without it being called such.

NITIN MANGOTRA

6.2 Why do we use '____init____()' Method In Python?

____init____() Method

- ❑ The ____init____ method is similar to **constructors** in C++ and Java. Constructors are used to initialize the object's state.
- ❑ It is run as soon as an object of a class is instantiated.
- ❑ The method is useful to do any initialization you want to do with your object.

For Example:

A Sample class with init method
class Person:

init method or constructor
def ____init____(self, name):
 self.name = name

Sample Method
def say_hi(self):
 print('Hello, my name is', self.name)

p = Person('Nitin')
p.say_hi()

Output:
Hello, my name is Nitin

6.2 Why do we use '____init____()' Method In Python?

```
# A Sample class with init method
class Person:
    # init method or constructor
    def __init__(self, name):
        self.name = name

    # Sample Method
    def say_hi(self):
        print('Hello, my name is', self.name)

p = Person('Nitin')
p.say_hi()

p2 = Person('Pranshul')
p2.say_hi()

p3 = Person('Utkarsh')
p3.say_hi()
```

Output:

```
Hello, my name is Nitin
Hello, my name is Pranshul
Hello, my name is Utkarsh
```

6.3 What is 'init' Keyword In Python?

__init__.py file

- ❑ The __init__.py file lets the Python interpreter know that a directory contains code for a Python module.
- ❑ It can be blank.
- ❑ Without one, you cannot import modules from another folder into your project.
- ❑ The role of the __init__.py file is similar to the __init__ function in a Python class.
- ❑ The file essentially the constructor of your package or directory without it being called such

__init__() function

- ❑ The __init__ method is similar to constructors in C++ and Java. Constructors are used to initialize the object's state.
- ❑ It is run as soon as an object of a class is instantiated.
- ❑ The method is useful to do any initialization you want to do with your object.

A Sample class with init method

class Person:

init method or constructor

def __init__(self, name):

self.name = name

Sample Method

def say_hi(self):

print('Hello, my name is', self.name)

p = Person('Nitin')

p.say_hi()

Output:

Hello, my name is Nitin

7.1 What Do You Mean By Module in Python

Module

- ❑ The module is a simple Python file that contains collections of functions and global variables and with having a .py extension file.
- ❑ A module is a single file (or files) that are imported under one import and are used.
- ❑ It is an executable file and to organize all the modules we have the concept called **Package** in Python.

E.g.

```
import <my_module>
```

```
Import json
```


7.2 What Do You Mean By Package in Python

Package

- ❑ The package is a simple directory having collections of modules.
- ❑ This directory contains Python modules and also having `__init__.py` file by which the interpreter interprets it as a Package.
- ❑ The package is simply a namespace.
- ❑ The package also contains sub-packages inside it.

A package is a collection of modules in directories that give a package hierarchy.

E.g

```
from my_package.abc import a
```

7.3 What Do You Mean By Library in Python

Library

- ❑ A library is an umbrella term referring to a reusable chunk of code.
- ❑ A Package is a collection of modules, a library is a collection of packages.
- ❑ Generally, A Python library contains a collection of related modules and packages.
- ❑ We create Python libraries to share reusable code with the community.

E.g

```
import <my_library>
```

```
Import numpy
```

7.4 Difference Between Modules and Packages in Python

Module

- ❑ The module is a simple Python file that contains collections of functions and global variables and with having a .py extension file.
- ❑ A module is a single file (or files) that are imported under one import and are used.
- ❑ It is an executable file and to organize all the modules we have the concept called **Package** in Python.

E.g.

```
import <my_module>
```

```
Import json
```

Package

- ❑ The package is a simple directory having collections of modules.
- ❑ This directory contains Python modules and also having `__init__.py` file by which the interpreter interprets it as a Package.
- ❑ The package is simply a namespace.
- ❑ The package also contains sub-packages inside it.
- ❑ A package is a collection of modules in directories that give a package hierarchy.

E.g

```
from my_package.abc import a
```

7.5 Difference Between Modules and Packages in Python

Module

- ❑ The module is a simple Python file that contains collections of functions and global variables and with having a .py extension file.
- ❑ A module is a single file (or files) that are imported under one import and are used.
- ❑ It is an executable file and to organize all the modules we have the concept called **Package** in Python.

E.g.

```
import <my_module>
```

```
Import json
```

Package

- ❑ The package is a simple directory having collections of modules.
- ❑ This directory contains Python modules and also having `__init__.py` file by which the interpreter interprets it as a Package.
- ❑ The package is simply a namespace.
- ❑ The package also contains sub-packages inside it.
- ❑ A package is a collection of modules in directories that give a package hierarchy.

E.g

```
from my_package.abc import a
```

Library

- ❑ A library is an umbrella term referring to a reusable chunk of code.
- ❑ A Package is a collection of modules, a library is a collection of packages.
- ❑ Generally, A Python library contains a collection of related modules and packages.
- ❑ We create Python libraries to share reusable code with the community.

E.g

```
import <my_library>
```

```
Import numpy
```

7.6 Give Some Examples Of Modules, Packages & Library In Python

Example Of Module

- ☐ random
- ☐ os
- ☐ json
- ☐ html
- ☐ datetime
- ☐ re

Example Of Package

- ☐ NumPy
- ☐ pandas
- ☐ pytest
- ☐ SciPy
- ☐ PyTorch
- ☐ Pendulum
- ☐ Pywin32

Example Of Module

- ☐ Matplotlib
- ☐ pygame
- ☐ Beautiful Soup
- ☐ Requests

8. What Is Range And Xrange In Python?

- ❑ The range() and xrange() are two functions that could be used to iterate a certain number of times in for loops in Python.
- ❑ In Python 3, there is no xrange, but the range function behaves like xrange in Python 2.
- ❑ range() – This returns a range object (a type of iterable).
- ❑ xrange() – This function returns the generator object that can be used to display numbers only by looping. The only particular range is displayed on demand and hence called “lazy evaluation”.

```
a = range(1,5000)
b = xrange(1,5000)

print ("The return type of range() is : ")
print (type(a))

print ("The return type of xrange() is : ")
print (type(b))
```

```
The return type of range() is :
<type 'list'>
The return type of xrange() is :
<type 'xrange'>
```

8.1 Difference Between Range and Xrange?

Parameters	Range()	Xrange()
Return type	It returns a list of integers.	It returns a generator object.
Memory Consumption	Since range() returns a list of elements, it takes more memory.	In comparison to range(), it takes less memory.
Speed	Its execution speed is slower.	Its execution speed is faster.
Python Version	Python 2, Python 3	xrange no longer exists.
Operations	Since it returns a list, all kinds of arithmetic operations can be performed.	Such operations cannot be performed on xrange().

8.2 The range() In Python 2 Is In Which Form In Python 3?

- ❑ The `range()` we use in Python 3 doesn't exist in Python 2.
- ❑ But the `xrange()` that we use in Python 2 is in the form of `range()` in Python 3.

```
main.py × [Menu] Console Shell
1 #python 3
2 a = range(1,10000)
3
4 print ("The return type of range() is : ")
5 print (type(a))
```

The return type of range() is :
<class 'range'>

8.3 On The Basis Of Memory, Which Is Better Range Or Xrange?

On The Basis Of Memory Consumption, **xrange** is better than **range()**.

Reason :-

- ❑ The variable storing the range created by **range()** takes more memory as compared to the variable storing the range using **xrange()**.
- ❑ The basic reason for this is the return type of **range()** is list and **xrange()** is **xrange()** object.

```
import sys

a = range(1,5000)
b = xrange(1,5000)

print ("The Memory Consumption using range() is : ")
print (sys.getsizeof(a))

print ("The Memory Consumption using xrange() is : ")
print (sys.getsizeof(b))
```

```
The Memory Consumption using range() is :
40064

The Memory Consumption using xrange() is :
40
```

8.4 On The Basis Of Execution Speed, Which Is Better Range Or Xrange?

On The Basis Of Execution Speed, xrange is better than range().

Reason :-

- ❑ Because of the fact that xrange() evaluates only the generator object containing only the values that are required by lazy evaluation,
- ❑ Therefore xrange() faster in implementation than range().

Execution Speed using range is :

0:00:00.004683

Execution Speed using xrange is :

0:00:00.004030

```
from datetime import datetime
a = range(1,50000)
b = xrange(1,50000)

#for range()
t1 = datetime.now()
for i in a:
    var = i
t2 = datetime.now()

print ("Execution Speed using range is : ")
print (t2-t1)

#For xrange()
t1 = datetime.now()
for i in b:
    var = i
t2 = datetime.now()

print ("Execution Speed using xrange is : ")
print (t2-t1)
```

8.5 On The Basis Of Operations Performed, Which Is Better Range Or Xrange?

On The Basis Of Operations Performed, **range** is better than **xrange()**.

Reason :-

- ❑ As `range()` returns the list, all the operations that can be applied on the list can be used on it.
- ❑ On the other hand, as `xrange()` returns the `xrange` object, operations associated to list cannot be applied on them, hence a disadvantage.

```
a = range(1,5000)
b = xrange(1,5000)

print ("The list after slicing using range is : ")
print (a[200:210])

print ("The list after slicing using xrange is : ")
print (b[200:210])
```

```
The list after slicing using range is :
[201, 202, 203, 204, 205, 206, 207, 208, 209, 210]
The list after slicing using xrange is :

Traceback (most recent call last):
  File "jdoodle.py", line 8, in <module>
    print (b[200:210])
TypeError: sequence index must be integer, not 'slice'
```

9.1 What are Generators In Python. Explain with Example.

- A generator is a special type of function which does not return a single value, instead, it returns an iterator object with a sequence of values.
- In a generator function, a “yield” statement is used rather than a return statement. I.e Generator uses “yield” keyword.
- Generators are iterators which can execute only once.
- Generators are mostly used in loops to generate an iterator by returning all the values in the loop without affecting the iteration of the loop

Example:

```
def sqr(n):  
    for i in range(1, n+1):  
        yield i*i  
a = sqr(3)
```

```
print("The square are : ")  
print(next(a))  
print(next(a))  
print(next(a))
```

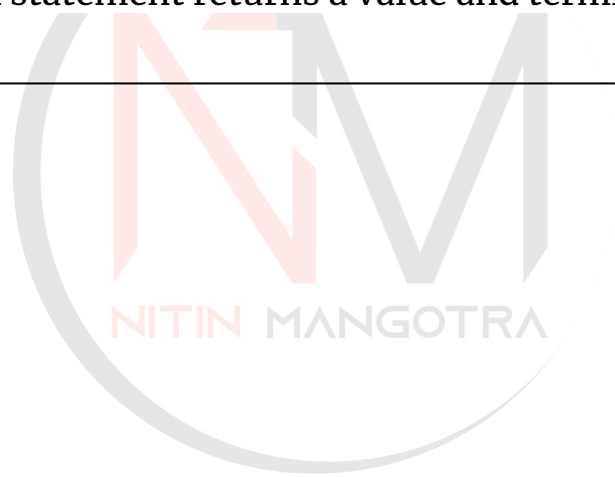
Output:

```
The square are :  
1  
4  
9
```

9.2 Diff Btw YIELD and RETURN Keywords In Python.

The difference between YIELD and RETURN is that :

- ❑ Yield returns a value and pauses the execution while maintaining the internal states,
- ❑ Whereas the return statement returns a value and terminates the execution of the function.



9.2 Diff Btw YIELD and RETURN Keywords In Python.

A Generator Example:

```
def generator1():  
    print('Hello NitMan')  
    yield 5  
  
    print('Hello Pranshul')  
    yield 10  
  
    print('Hello Utkarsh')  
    yield 15
```

```
generator_values = generator1()  
print(generator_values)
```

```
print(next(generator_values))  
print(next(generator_values))  
print(next(generator_values))
```

Console Shell

```
<generator object generator1 at 0x7fec0419d350>  
Hello NitMan  
5  
Hello Pranshul  
10  
Hello Utkarsh  
15  
✖ □
```

In the above example, the mygenerator() function is a generator function.

It uses yield instead of return keyword. So, this will return the value against the yield keyword each time it is called.

9.3 Explain Different Ways To Access Element In A Generator.

1. By Using next() Function

```
main.py x
1 def generator1():
2     print('Hello NitMan')
3     yield 5
4
5     print('Hello Pranshul')
6     yield 10
7
8     print('Hello Utkarsh')
9     yield 15
10
11 generator_values = generator1()
12 print(generator_values)
13
14 print(next(generator_values))
15 print(next(generator_values))
16 print(next(generator_values))
```

Console Shell

```
<generator object generator1 at 0x7fec0419d350>
Hello NitMan
5
Hello Pranshul
10
Hello Utkarsh
15
>
```

9.3 Explain Different Ways To Access Element In A Generator.

2. By Using for loop

```
main.py × ☰ Console Shell
1 ▼ def generator1():
2     print('Hello NitMan')
3     yield 5
4
5     print('Hello Pranshu')
6     yield 10
7
8     print('Hello Utkarsh')
9     yield 15
10
11 generator_values = generator1()
12 ▼ for i in generator_values:
13     print(i)
```

Hello NitMan
 5
 Hello Pranshu
 10
 Hello Utkarsh
 15
 ✨

9.4 What Happen When We Use RETURN Before & After YIELD Keyword In A Generator In Python.

As we know, “YIELD” returns a value and pauses the execution while maintaining the internal states, whereas the “RETURN” statement returns a value and terminates the execution of the function.

- ❑ When We Use “RETURN” before “YIELD” keyword, the function won’t execute the remaining “YIELD” Statements.
- ❑ And If We try to access the next Yield element, It will throw an error “ STOP ITERATION”.
- ❑ When We Use “RETURN” after “YIELD” Keyword, the function first execute the “YIELD” statement, and then it will execute “RETURN” statement.

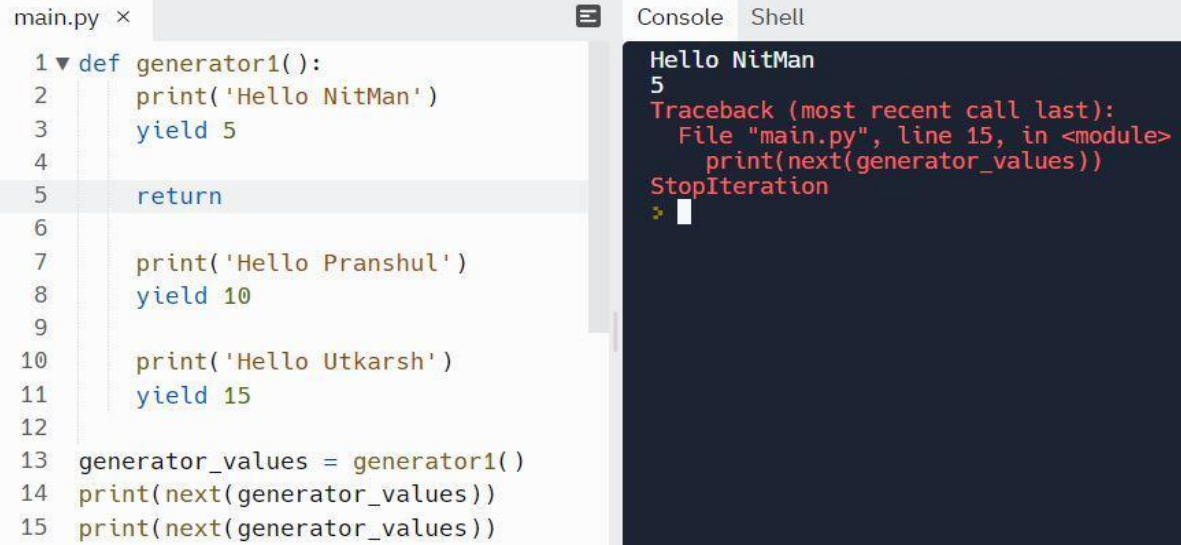
NOTE: After “RETURN” statement, A value returns and terminates the execution of the function.

9.4 What Happen When We Use RETURN Before & After YIELD Keyword In A Generator In Python.

- ❑ When We Use “RETURN” before “YIELD”, the function won’t execute the remaining “YIELD” Statements.
- ❑ And If We try to access the next Yield element, It will throw an error “ STOP ITERATION”.
- ❑ When We Use “RETURN” after “YIELD”, the function first execute the “YIELD” statement, and then it will execute “RETURN”.

Explanation:

As you can see, the generator (**generator1()**) stops executing after getting the first item because the return keyword is used after yielding the first item.



```

main.py x
1 def generator1():
2     print('Hello NitMan')
3     yield 5
4
5     return
6
7     print('Hello Pranshul')
8     yield 10
9
10    print('Hello Utkarsh')
11    yield 15
12
13 generator_values = generator1()
14 print(next(generator_values))
15 print(next(generator_values))
  
```

```

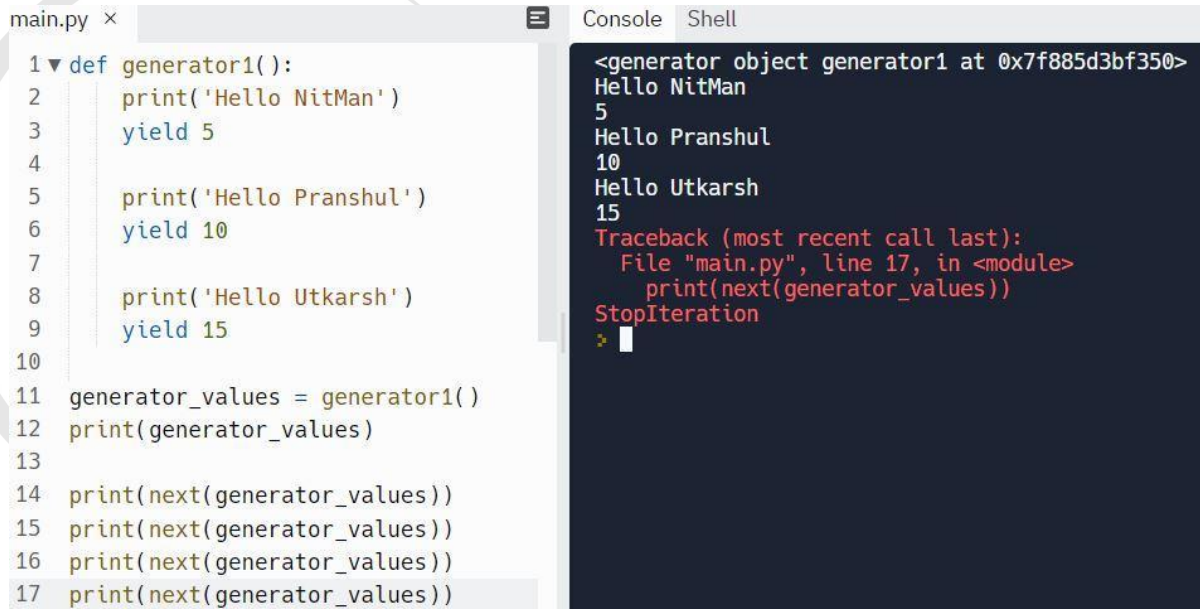
Hello NitMan
5
Traceback (most recent call last):
  File "main.py", line 15, in <module>
    print(next(generator_values))
StopIteration
  
```

9.5 What Happen When We Try To Access An Element That Is Not In Range OF A Generator?

- ❑ We we get an error of “ STOP ITERATION”.

Explanation:

As you can see, the generator (**generator1()**) stops executing after getting the first item because the return keyword is used after yielding the first item.



```

main.py x
1 def generator1():
2     print('Hello NitMan')
3     yield 5
4
5     print('Hello Pranshul')
6     yield 10
7
8     print('Hello Utkarsh')
9     yield 15
10
11 generator_values = generator1()
12 print(generator_values)
13
14 print(next(generator_values))
15 print(next(generator_values))
16 print(next(generator_values))
17 print(next(generator_values))

Console Shell
<generator object generator1 at 0x7f885d3bf350>
Hello NitMan
5
Hello Pranshul
10
Hello Utkarsh
15
Traceback (most recent call last):
  File "main.py", line 17, in <module>
    print(next(generator_values))
StopIteration
>
    
```

10.1 What are in-built Data Types in Python.

DataType	Mutable Or Immutable?
Boolean (bool)	Immutable
Integer (int)	Immutable
Float	Immutable
String (str)	Immutable
tuple	Immutable
frozenset	Immutable
list	Mutable
set	Mutable
dict	Mutable

10.2 Difference Between Mutable and Immutable Data Types

- ❑ A first fundamental distinction that Python makes on data is about whether or not the value of an object changes.
- ❑ If the value can change, the object is called **Mutable**,
- ❑ while if the value cannot change, the object is called **Immutable**.

DataType	Mutable Or Immutable?
Boolean (bool)	Immutable
Integer (int)	Immutable
Float	Immutable
String (str)	Immutable
tuple	Immutable
frozenset	Immutable
list	Mutable
set	Mutable
dict	Mutable

11.1 What Is Ternary Operator. Explain It With An Example.

- ❑ The ternary operator in python is used to return a value based on the result of a binary condition.
- ❑ It takes binary value(condition) as an input, so it looks similar to an “if-else” condition block.
- ❑ However, it also returns a value so behaving similar to a function.

The syntax for the Python Ternary statement is as follows:

[If True, first Value] if [expression] else [If False Second_Value]

Example:

```
result= 30 if (5 > 6) else 40
> print(result)
Output > 40
```

```
passing_marks = 33
your_marks = 23
result= 'Bike' if (your_marks > passing_marks) else 'Chhapal'
print(result)
Output > Chhapal
```

11.2 Compare The Ternary Operator In Java & Python.

In Case Of Java

The syntax for the Java Ternary Operator is as follows:

`(expression) ? (If True, first Value) : (If False Sec_Value)`

Ternary Operator Example:

```
result = (5>6) ? 30 : 40  
print(result)  
⇒ 40
```

In Case Of Python

The syntax for the Python Ternary Operator is as follows:

`[If True, first Value] if [expression] else [If False Sec_Value]`

Ternary Operator Example:

```
result= 30 if (5 > 6) else 40  
print(result)  
40
```

11.3 Find The Maximum Value By Using The Ternary Operator.

```
a = 10
```

```
b = 20
```

```
c = 30
```

```
max_value = a if a>b>c else b if b>c else c
```

```
print(max_value)
```

```
⇒ 30
```


18. Difference Between Generators And Iterators

GENERATOR

1. Generators are mostly used in loops to generate an iterator by returning all the values in the loop without affecting the iteration of the loop.
2. Generator uses “**yield**” keyword.
3. Every Generator is an iterator
4. Generators in Python are more memory efficient.
5. Generators are iterators which can execute only once.

ITERATOR

1. An iterator is an object which contains a countable number of values and it is used to iterate over iterable objects like list, tuples, sets, etc.
2. Iterator uses `iter()` and `next()` functions.
3. Every iterator is not a generator.
4. Iterators in python are less memory efficient.
5. Iterators are used mostly to iterate or convert other objects to an iterator using `iter()` function.

One of the advantages of the generator over the iterator is that elements are generated dynamically. Since the next item is generated only after the first is consumed, it is more memory efficient than the iterator.

Thanks! Hope It Helps You!

Watch The Answers For The Remaining Questions On My Youtube Channel.

Link For The Remaining Questions :
<https://youtube.com/playlist?list=PL1B8DuzqTNOJRjMrCrVdoa1WKyicperOb>

Connect with me:

Youtube: <https://www.youtube.com/c/nitmantalks>

Instagram: <https://www.instagram.com/nitinmangotra/>

LinkedIn: <https://www.linkedin.com/in/nitin-mangotra-9a075a149/>

Facebook: <https://www.facebook.com/NitManTalks/>

Twitter: <https://twitter.com/nitinmangotra07/>

Telegram: <https://t.me/nitmantalks/>

Do Connect With Me!!!!

Please Do Comment Your Feedback In Comment Section Of My Video On Youtube.
Here Is The Link:

<https://youtube.com/playlist?list=PL1B8DuzqTNOJRjMrCrVd0a1WKyicperOb>

When You Get Placed In Any Company Because Of My Video, DO Let Me Know.
It will Give Me More Satisfaction and Will Motivate me to make more such video Content!!

Thanks

PS: Don't Forget To Connect With ME.

Regards,

Nitin Mangotra (NitMan)

Connect with me:

Youtube: <https://www.youtube.com/c/nitmantalks>

Instagram: <https://www.instagram.com/nitinmangotra/>

LinkedIn: <https://www.linkedin.com/in/nitin-mangotra-9a075a149/>

Facebook: <https://www.facebook.com/NitManTalks/>

Twitter: <https://twitter.com/nitinmangotra07/>

Telegram: <https://t.me/nitmantalks/>