

1. Difference Between List and Tuple

1

LIST

1. Lists are **mutable**
2. List is a container to contain different types of objects and is used to iterate objects.
3. Syntax Of List

```
list = ['a', 'b', 'c', 1, 2, 3]
```
4. List iteration is slower
5. Lists consume more memory
6. Operations like insertion and deletion are better performed.

Tuple

1. Tuples are **immutable**
2. Tuple is also similar to list but contains immutable objects.
3. Syntax Of Tuple

```
tuples = ('a', 'b', 'c', 1, 2)
```
4. Tuple processing is faster than List.
5. Tuple consume less memory
6. Elements can be accessed better.

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. What is Decorator? Explain With Example.

2

```
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()
```

NITIN MANGOTRA

3. Difference Between List and Dict Comprehension

List Comprehension

Example:
Common Way:

```
l = []  
for i in range(10):  
    if i%2:  
        l.append(i)  
print(l)
```

Dict Comprehension

Example:
Common Way:

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

NITIN MANGOTRA

3. Difference Between List and Dict Comprehension

3

List Comprehension

Example:

Common Way:

```
l = []
for i in range(10):
    if i%2:
        l.append(i)
print(l)
```

Dict Comprehension

Example:

Common Way:

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

NITIN MANGOTRA

3. Difference Between List and Dict Comprehension

List Comprehension

Using List Comprehension:

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

Output:

[1, 3, 5, 7, 9]

Dict Comprehension

Using Dict Comprehension:

```
d1={n:n*n for n 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}

NITIN MANGOTRA

4. How Memory Managed In Python?

4

- 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.



5. Difference Between Generators And Iterators

GENERATOR

- Generators are iterators which can execute only once.
- Generator uses “`yield`” keyword.
- 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.
- Every Generator is an iterator

ITERATOR

- 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.
- Every iterator is not a generator.

5. Difference Between Generators And Iterators

GENERATOR

EXAMPLE:

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

Output:

```
1
4
9
```

ITERATOR

Example:

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

Output:

```
A
B
C
```



5. Difference Between Generators And Iterators

GENERATOR

- Generators are iterators which can execute only once.
- Generator uses "yield" keyword.
- Generators are mostly used in loop to generate an iterator by returning all the values in the loop without affecting the iteration of the loop.
- Every generator is an iterator.

EXAMPLE:

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

Output:

```
1
4
9
```

ITERATOR

- 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.
- 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
```



6. 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. It sets up how packages or functions will be imported into your other files.

`__init__()` function

The `__init__` method is similar to **constructors** in C++ and Java. Constructors are used to initialize the object's state.

```
# 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()
```



7. Difference Between Modules and Packages in Python

Module

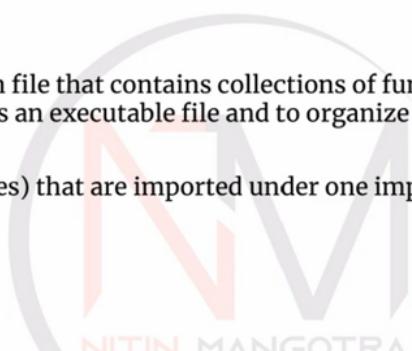
The module is a simple Python file that contains collections of functions and global variables and with having a `.py` extension file. It is an executable file and to organize all the modules we have the concept called **Package** in Python.

A module is a single file (or files) that are imported under one import and used.

E.g.

```
import <my_module>
```

```
Import numpy
```



7. Difference Between Modules and Packages in Python

7

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

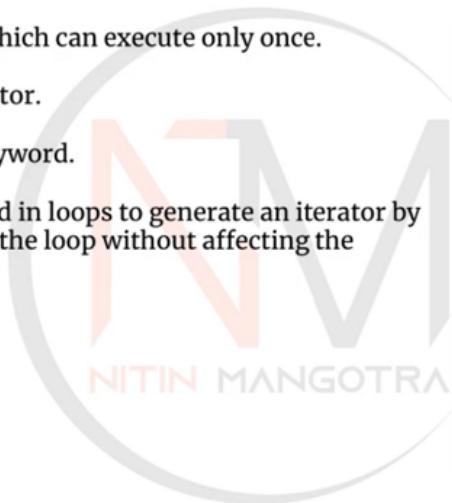
```
from my_package.abc import a
```

8. Difference Between Range and Xrange?

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

9. What are Generators. Explain it with Example.

- Generators are iterators which can execute only once.
- Every generator is an iterator.
- Generator uses “yield” keyword.
- 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))
```

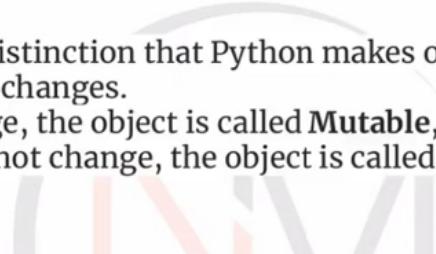
Output:

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

10. What are in-built Data Types in Python OR Explain 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**.



10. What are in-built Data Types in Python OR Explain Mutable and Immutable Data Types

9

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. Explain Ternary Operator in Python?

The syntax for the Python ternary statement is as follows:

[if_true] if [expression] else [if_false]

Ternary Operator Example:

```
age = 25
discount = 5 if age < 65 else 10
print(discount)
```

12. What is Inheritance In Python

In **Inheritance**, the child class acquires the properties and can access all the data members and functions defined in the parent class. A child class can also provide its specific implementation to the functions of the parent class.

In python, a derived class can inherit base class by just mentioning the base in the bracket after the derived class name.

Class A(B):

12. What is Inheritance In Python

Example:

```
class A:  
    def display(self):  
        print("A Display")
```

```
class B(A):  
    def show(self):  
        print("B Show")
```

```
d = B()  
d.show()  
d.display()
```

Output:
B Show
A Display



13. Difference Between Local and Global Variable in Python

11

Local Variable	Global Variable
It is declared inside a function.	It is declared outside the function.
If it is not initialized, a garbage value is stored	If it is not initialized zero is stored as default.
It is created when the function starts execution and lost when the functions terminate.	It is created before the program's global execution starts and lost when the program terminates.
Data sharing is not possible as data of the local variable can be accessed by only one function.	Data sharing is possible as multiple functions can access the same global variable.
Parameters passing is required for local variables to access the value in other function	Parameters passing is not necessary for a global variable as it is visible throughout the program
When the value of the local variable is modified in one function, the changes are not visible in another function.	When the value of the global variable is modified in one function changes are visible in the rest of the program.
Local variables can be accessed with the help of statements, inside a function in which they are declared.	You can access global variables by any statement in the program.
It is stored on the stack unless specified.	It is stored on a fixed location decided by the compiler.



14. Explain Break, Continue and Pass Statement

- ▶ A **break** statement, when used inside the loop, will terminate the loop and exit. If used inside nested loops, it will break out from the current loop.
- ▶ A **continue** statement will stop the current execution when used inside a loop, and the control will go back to the start of the loop.
- ▶ A **pass** statement is a null statement. When the Python interpreter comes across the pass statement, it does nothing and is ignored.

14. Explain Break, Continue and Pass Statement

12

Break Statement Example

```
for i in range(10):  
    if i == 7:  
        break  
    print(i, end = ",")
```

Output:
0,1,2,3,4,5,6,

Continue Statement Example

```
for i in range(10):  
    if i == 7:  
        continue  
    print(i, end = ",")
```

Output:
0,1,2,3,4,5,6,8,9,

Pass Statement Example

```
def my_func():  
    print('pass inside function')  
    pass  
my_func()
```

Output:
pass inside function

15. What is 'self' Keyword in python?

The 'self' parameter is a reference to the current instance of the class, and is used to access variables that belongs to the class.

```
class Person:  
    def __init__(self, name, age):  
        self.name = name  
        self.age = age  
  
    def info(self):  
        print(f"My name is {self.name}. I am {self.age} years old.")  
  
c = Person("Nitin", 23)  
c.info()
```

Output:
My name is Nitin. I am 23 years old.



16. Difference Between Pickling and Unpickling?

13

Pickling:

In python, the pickle module accepts any Python object, transforms it into a string representation, and dumps it into a file by using the dump function. This process is known as **pickling**. The function used for this process is pickle.dump()

Unpickling:

The process of retrieving the original python object from the stored string representation is called **unpickling**. The function used for this process is pickle.load()

- They are inverses of each other.
- **Pickling**, also called **serialization**, involves converting a Python object into a series of bytes which can be written out to a file.
- **Unpickling**, or **de-serialization**, does the opposite—it converts a series of bytes into the **Python object** it represents.

17. Explain Function of List, Set, Tuple And Dictionary?

Functions Of List

- sort(): Sorts the list in ascending order.
- append(): Adds a single element to a list.
- extend(): Adds multiple elements to a list.
- index(): Returns the first appearance of the specified value.
- max(list): It returns an item from the list with max value.
- min(list): It returns an item from the list with min value.
- len(list): It gives the total length of the list.
- list(seq): Converts a tuple into a list.
- cmp(list1, list2): It compares elements of both lists list1 and list2.
- type(list): It returns the class type of an object.

Functions Of Tuple

- cmp(tuple1, tuple2): Compares elements of both tuples.
- len(tuple): Returns the length of the tuple.
- max(): Returns item from the tuple with max value.
- min(): Returns item from the tuple with min value.
- tuple(seq): Converts a list into tuple.
- sum(): returns the arithmetic sum of all the items in the tuple.
- any(): If even one item in the tuple has a Boolean value of True, it returns True. Otherwise, it returns False.
- all(): returns True only if all items have a Boolean value of True. Otherwise, it returns False.
- sorted(): a sorted version of the tuple.
- index(): It takes one argument and returns the index of the first appearance of an item in a tuple.
- count(): It takes one argument and returns the number of times an item appears in the tuple.

Functions Of Dictionary

- clear(): Removes all the elements from the dictionary
- copy(): Returns a copy of the dictionary
- fromkeys(): Returns a dictionary with the specified keys and value
- get(): Returns the value of the specified key
- items(): Returns a list containing a tuple for each key value pair
- keys(): Returns a list containing the dictionary's keys
- pop(): Removes the element with the specified key
- popitem(): Removes the last inserted key-value pair
- setdefault(): Returns the value of the specified key. If the key does not exist: insert the key, with the specified value
- update(): Updates the dictionary with the specified key-value pairs
- values(): Returns a list of all the values in the dictionary
- cmp(): compare two dictionaries

Functions Of Set

- add(): Adds an element to the set
- clear(): Removes all the elements from the set
- copy(): Returns a copy of the set
- difference(): Returns a set containing the difference between two or more sets
- difference_update(): Removes the items in this set that are also included in another, specified set
- discard(): Remove the specified item
- intersection(): Returns a set, that is the intersection of two or more sets
- intersection_update(): Removes the items in this set that are not present in other, specified set(s)
- isdisjoint(): Returns whether two sets have a intersection or not
- issubset(): Returns whether another set contains this set or not
- issuperset(): Returns whether this set contains another set or not
- pop(): Returns an element from the set
- remove(): Removes the specified element
- symmetric_difference(): Returns a set with the symmetric differences of two sets
- symmetric_difference_update(): inserts the symmetric differences from this set and another
- union(): Return a set containing the union of sets
- update(): Update the set with another set, or any other iterable

17. Explain Function of List, Set, Tuple And Dictionary?

14

Functions Of List

- ❑ `sort()`: Sorts the list in ascending order.
- ❑ `append()`: Adds a single element to a list.
- ❑ `extend()`: Adds multiple elements to a list.
- ❑ `index()`: Returns the first appearance of the specified value.
- ❑ `max(list)`: It returns an item from the list with max value.
- ❑ `min(list)`: It returns an item from the list with min value.
- ❑ `len(list)`: It gives the total length of the list.
- ❑ `list(seq)`: Converts a tuple into a list.
- ❑ `cmp(list1, list2)`: It compares elements of both lists list1 and list2.
- ❑ `type(list)`: It returns the class type of an object.

17. Explain Function of List, Set, Tuple And Dictionary?

Functions Of Tuple

- ❑ `cmp(tuple1, tuple2)`: Compares elements of both tuples.
- ❑ `len()`: total length of the tuple.
- ❑ `max()`: Returns item from the tuple with max value.
- ❑ `min()`: Returns item from the tuple with min value.
- ❑ `tuple(seq)`: Converts a list into tuple.
- ❑ `sum()`: returns the arithmetic sum of all the items in the tuple.
- ❑ `any()`: If even one item in the tuple has a Boolean value of True, it returns True. Otherwise, it returns False.
- ❑ `all()`: returns True only if all items have a Boolean value of True. Otherwise, it returns False.
- ❑ `sorted()`: a sorted version of the tuple.
- ❑ `index()`: It takes one argument and returns the index of the first appearance of an item in a tuple
- ❑ `count()`: It takes one argument and returns the number of times an item appears in the tuple.

17. Explain Function of List, Set, Tuple And Dictionary?

15

Functions Of Dictionary

- `clear()`: Removes all the elements from the dictionary
- `copy()`: Returns a copy of the dictionary
- `fromkeys()`: Returns a dictionary with the specified keys and value
- `get()`: Returns the value of the specified key
- `items()`: Returns a list containing a tuple for each key value pair
- `keys()`: Returns a list containing the dictionary's keys
- `pop()`: Removes the element with the specified key
- `popitem()`: Removes the last inserted key-value pair
- `setdefault()`: Returns the value of the specified key. If the key does not exist: insert the key, with the specified value
- `update()`: Updates the dictionary with the specified key-value pairs
- `values()`: Returns a list of all the values in the dictionary
- `cmp()`: compare two dictionaries

17. Explain Function of List, Set, Tuple And Dictionary?

Functions Of Set

- `add()`: Adds an element to the set
- `clear()`: Removes all the elements from the set
- `copy()`: Returns a copy of the set
- `difference()`: Returns a set containing the difference between two or more sets
- `difference_update()`: Removes the items in this set that are also included in another, specified set
- `discard()`: Remove the specified item
- `intersection()`: Returns a set, that is the intersection of two or more sets
- `intersection_update()`: Removes the items in this set that are not present in other, specified set(s)
- `isdisjoint()`: Returns whether two sets have a intersection or not
- `issubset()`: Returns whether another set contains this set or not
- `issuperset()`: Returns whether this set contains another set or not
- `pop()`: Removes an element from the set
- `remove()`: Removes the specified element
- `symmetric_difference()`: Returns a set with the symmetric differences of two sets
- `symmetric_difference_update()`: inserts the symmetric differences from this set and another
- `union()`: Return a set containing the union of sets
- `update()`: Update the set with another set, or any other iterable



17. Explain Function of List, Set, Tuple And Dictionary?

16

Functions Of Set

- `add()`: Adds an element to the set
- `clear()`: Removes all the elements from the set
- `copy()`: Returns a copy of the set
- `difference()`: Returns a set containing the difference between two or more sets
- `difference_update()`: Removes the items in this set that are also included in another, specified set
- `discard()`: Remove the specified item
- `intersection()`: Returns a set, that is the intersection of two or more sets
- `intersection_update()`: Removes the items in this set that are not present in other, specified set(s)
- `isdisjoint()`: Returns whether two sets have a intersection or not
- `issubset()`: Returns whether another set contains this set or not
- `issuperset()`: Returns whether this set contains another set or not
- `pop()`: Removes an element from the set
- `remove()`: Removes the specified element
- `symmetric_difference()`: Returns a set with the symmetric differences of two sets
- `symmetric_difference_update()`: inserts the symmetric differences from this set and another
- `union()`: Return a set containing the union of sets
- `update()`: Update the set with another set, or any other iterable

19. Explain Type Conversion in Python. [(`int()`, `float()`, `ord()`, `oct()`, `str()` etc.)]

- **`int()`** - Converts any data type into an integer.

Example:

```
a = '100'  
d = int(a)  
print(d)  
print(type(d))
```

Output:

```
100  
<class 'int'>
```

19. Explain Type Conversion in Python. [(int(), float(), ord(), oct(), str() etc.)]

17

- **float()** - Returns A floating point number from a number or string

Example:

```
a = '100'  
d = float(a)  
print(d)  
print(type(d))
```

Output:

```
100.0  
<class 'float'>
```



- **oct()** - Returns its octal representation in a string format.

Example:

```
a = 100  
d = oct(a)  
print(d)  
print(type(d))
```

Output:

```
0o144  
<class 'str'>
```



- ❑ **hex()** - Convert the integer into a suitable hexadecimal form for the number of the integer.

18

Example:

```
a = 100
d = hex(a)
print(d)
print(type(d))
```

Output:

```
0x64
<class 'str'>
```



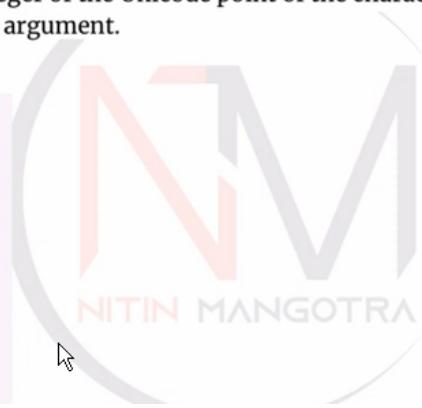
- ❑ **ord()** - Returns the integer of the Unicode point of the character in the Unicode case or the byte value in the case of an 8-bit argument.

Example:

```
a = 'A'
d = ord(a)
print(d)
print(type(d))
```

Output:

```
65
<class 'int'>
```



- ❑ **chr()** - Returns the character (string) from the integer (represents unicode code point of the character).

Example:

```
a = 100
d = chr(a)
print(d)
print(type(d))
```

Output:

```
d
<class 'str'>
```



- ❑ **eval()** - Parses the expression argument and evaluates it as a python expression.

19

Example:

```
a = '100+2+3'  
d = eval(a)  
print(d)  
print(type(d))
```

Output:

```
105  
<class 'int'>
```



- ❑ **str()** - Convert a value (integer or float) into a string.

Example:

```
a = 100  
d = str(a)  
print(d)  
print(type(d))
```

Output:

```
100  
<class 'str'>
```



- ❑ **repr()** - Returns the string representation of the value passed to eval function by default. For the custom class object, it returns a string enclosed in angle brackets that contains the name and address of the object by default.

20

Example:

```
a = 100
d = repr(a)
print(d)
print(type(d))
```

Output:

```
100
<class 'str'>
```



20. What does *args and **kwargs mean? Explain

When you are not clear how many arguments you need to pass to a particular function, then we use ***args** and ****kwargs**.

The ***args** keyword represents a varied number of arguments. It is used to add together the values of multiple arguments

The ****kwargs** keyword represents an arbitrary number of arguments that are passed to a function. ****kwargs** keywords are stored in a dictionary. You can access each item by referring to the keyword you associated with an argument when you passed the argument.

20. What does *args and **kwargs mean? Explain

***args Python Example:**

```
def sum(*args):
    total = 0
    for a in args:
        total = total + a
    print(total)
```

```
sum(1,2,3,4,5)
```

Output:

15

****Kwargs Python Example**

```
def show(**kwargs):
    print(kwargs)

show(A=1,B=2,C=3)
```

Output:

```
{'A': 1, 'B': 2, 'C': 3}
```

21. What is "Open" and "With" statement in Python

21

- Both Statements are used in case of file handling.
- With the “With” statement, you get better syntax and exceptions handling.

```
f = open("nitin.txt")
content = f.read()
print(content)
f.close()
```

```
with open("nitin.txt") as f:
    content = f.read()
    print(content)
```



22. Different Ways To Read And Write In A File In Python?

- ❑ **Read Only ('r')** : Open text file for reading. The handle is positioned at the beginning of the file. If the file does not exists, raises I/O error. This is also the default mode in which file is opened.
- ❑ **Read and Write ('r+')** : Open the file for reading and writing. The handle is positioned at the beginning of the file. Raises I/O error if the file does not exists.
- ❑ **Write Only ('w')** : Open the file for writing. For existing file, the data is truncated and over-written. The handle is positioned at the beginning of the file. Creates the file if the file does not exists
- ❑ **Write and Read ('w+')** : Open the file for reading and writing. For existing file, data is truncated and over-written. The handle is positioned at the beginning of the file.
- ❑ **Append Only ('a')** : Open the file for writing. The file is created if it does not exist. The handle is positioned at the end of the file. The data being written will be inserted at the end, after the existing data.
- ❑ **Append and Read ('a+')** : Open the file for reading and writing. The file is created if it does not exist. The handle is positioned at the end of the file. The data being written will be inserted at the end, after the existing data.
- ❑ **Text mode ('t')**: meaning \n characters will be translated to the host OS line endings when writing to a file, and back again when reading.
- ❑ **Exclusive creation ('x')**: File is created and opened for writing – but only if it doesn't already exist. Otherwise you get a FileExistsError.
- ❑ **Binary mode ('b')**: appended to the mode opens the file in binary mode, so there are also modes like 'rb', 'wb', and 'r+b'.

24. How Exception Handled In Python?

22

Try: This block will test the exceptional error to occur.

Except: Here you can handle the error.

Else: If there is no exception then this block will be executed.

Finally: Finally block always gets executed either exception is generated or not.

```
try:  
    # Some Code....!  
  
except:  
    # Optional Block  
    # Handling of exception (if required)  
  
else:  
    # Some code .....  
    # execute if no exception  
  
finally:  
    # Some code .....(always executed)
```

26. What is 'PIP' In Python

Python pip is the package manager for Python packages. We can use pip to install packages that do not come with Python.

The basic syntax of pip commands in command prompt is:

```
pip 'arguments'  
  
Pip install <package_name>
```

28. How to use F String and Format or Replacement Operator?

#How To Use f-string

```
name = 'Nitin'  
role = 'Python Developer'  
print(f"Hello, My name is {name} and I'm {role}")
```

Output:

Hello, My name is Nitin and I'm Python Developer

27. Where Python Is Used?

23

- Web Applications
- Desktop Applications
- Database Applications
- Networking Application
- Machine Learning
- Artificial Intelligence
- Data Analysis
- IOT Applications
- Games and many more...!

28. How to use F String and Format or Replacement Operator?

#How To Use f-string

```
name = 'Nitin'  
role = 'Python Developer'  
print(f"Hello, My name is {name} and I'm {role}")
```

Output:

Hello, My name is Nitin and I'm Python Developer

#How To Use format Operator

```
name = 'Nitin'  
role = 'Python Developer'  
print("Hello, My name is {} and I'm {}".format(name,role))
```

Output:

Hello, My name is Nitin and I'm Python Developer



29. How to Get List of all keys in a Dictionary?

24

Case - 1,2: Using List

```
dct = {'A': 1, 'B': 2, 'C': 3}  
all_keys = list(dct.keys())  
print(all_keys)
```

Shortcut for Above Code:
dct = {'A': 1, 'B': 2, 'C': 3}
all_keys = list(dct)
print(all_keys)

Output:
['A', 'B', 'C']



29. How to Get List of all keys in a Dictionary?

Case - 3,4: Using Iterable Unpacking Operator

```
d = {'A': 1, 'B': 2, 'C': 3}  
x = [*d.keys()]  
print(x)
```

Shortcut For Above Code:
d = {'A': 1, 'B': 2, 'C': 3}
x = [*d]
print(x)

Output:
['A', 'B', 'C']



29. How to Get List of all keys in a Dictionary?

25

Case - 6,7: Using Iterable Unpacking Operator

```
d = {'A': 1, 'B': 2, 'C': 3}
*x, = d.keys()
print(x)
```

Shortcut For Above Code:
d = {'A': 1, 'B': 2, 'C': 3}
*x, = d
print(x)

Output :
['A', 'B', 'C']



30. Difference Between Abstraction and Encapsulation.

Abstraction	Encapsulation
Abstraction works on the design level.	Encapsulation works on the application level.
Abstraction is implemented to hide unnecessary data and withdrawing relevant data.	Encapsulation is the mechanism of hiding the code and the data together from the outside world or misuse.
It highlights what the work of an object instead of how the object works is	It focuses on the inner details of how the object works. Modifications can be done later to the settings.
Abstraction focuses on outside viewing, for example, shifting the car.	Encapsulation focuses on internal working or inner viewing, for example, the production of the car.
Abstraction is supported in Java with the interface and the abstract class.	Encapsulation is supported using, e.g. public, private and secure access modification systems.
In a nutshell, abstraction is hiding implementation with the help of an interface and an abstract class.	In a nutshell, encapsulation is hiding the data with the help of getters and setters.

31. Does Python Support Multiple Inheritance. (Diamond Problem)

26

Yes, Python Supports Multiple Inheritance.

What Is Diamond Problem?

Java does not allow is multiple inheritance where one class can inherit properties from more than one class. It is known as the **diamond problem**.

31. Does Python Support Multiple Inheritance. (Diamond Problem)

```
class A
{
    public void display()
    {
        System.out.println("class A");
    }
}
```

```
class B extends A
{
    @Override
    public void display()
    {
        System.out.println("class B");
    }
}
```

```
class C extends A
{
    @Override
    public void display()
    {
        System.out.println("class C");
    }
}
```

```
//not supported in Java
public class D extends B,C
{
    public static void main(String args[])
    {
        D d = new D();
        //creates ambiguity which display() method to call
        d.display();
    }
}
```

In the above figure, we find that class D is trying to inherit from class B and class C, that



31. Does Python Support Multiple Inheritance. (Diamond Problem)

27

Yes, Python Supports Multiple Inheritance.

What Is Diamond Problem?

What Java does not allow is multiple inheritance where one class can inherit properties from more than one class. It is known as the diamond problem.

```
class B extends A
{
    @Override
    public void display()
    {
        System.out.println("class B");
    }
}
```

```
class C extends A
{
    @Override
    public void display()
    {
        System.out.println("class C");
    }
}
```

```
class A
{
    public void display()
    {
        System.out.println("class A");
    }
}
```

```
//not supported in Java
public class D extends B,C
{
    public static void main(String args[])
    {
        D d = new D();
        //creates a
        d.display();
    }
}
```



In the above figure, we find that class D is trying to inherit from class B and class C, that

31. Does Python Support Multiple Inheritance. (Diamond Problem)

Multiple Inheritance In Python:

```
class A:
    def abc(self):
        print("a")

class B(A):
    def abc(self):
        print("b")

class C(A):
    def abc(self):
        print("c")

class D(B,C):
    pass

d = D()
d.abc()
```

Output:
b



Empty List:

`a = []`

Empty Tuple:

`a = ()`

Empty Dict:

`a = {}`

Empty Set:

`a = set()`

33. Difference Between .py and .pyc

- ❑ .py files contain the source code of a program. Whereas, .pyc file contains the bytecode of your program.
- ❑ Python compiles the .py files and saves it as .pyc files , so it can reference them in subsequent invocations.
- ❑ The .pyc contain the compiled bytecode of Python source files. This code is then executed by Python's virtual machine .

34. How Slicing Works In String Manipulation. Explain.

29

Syntax: Str_Object[Start_Position:End_Position:Step]

```
s = 'HelloWorld'
```

Indexing:

H	E	L	L	O	W	O	R	L	D
0	1	2	3	4	5	6	7	8	9
-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

```
print(s[:])
```

Output:
HelloWorld



34. How Slicing Works In String Manipulation. Explain

Syntax: Str_Object[Start_Position:End_Position:Step]

```
s = 'HelloWorld'
```

Indexing:

H	E	L	L	O	W	O	R	L	D
0	1	2	3	4	5	6	7	8	9
-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

```
print(s[:5])
```

Output:
Hello



34. How Slicing Works In String Manipulation. Explain.

30

Syntax: Str_Object[Start_Position:End_Position:Step]

```
s = 'HelloWorld'
```

Indexing:

H	E	L	L	O	W	O	R	L	D
0	1	2	3	4	5	6	7	8	9
-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

```
print(s[2:5])
```

Output:
llo



34. How Slicing Works In String Manipulation. Explain.

Syntax: Str_Object[Start_Position:End_Position:Step]

```
s = 'HelloWorld'
```

Indexing:

H	E	L	L	O	W	O	R	L	D
0	1	2	3	4	5	6	7	8	9
-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

```
print(s[2:8:2])
```

Output:
loo



34. How Slicing Works In String Manipulation. Explain.

31

Syntax: Str_Object[Start_Position:End_Position:Step]

```
s = 'HelloWorld'
```

Indexing:

H	E	L	L	O	W	O	R	L	D
0	1	2	3	4	5	6	7	8	9
-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

```
print(s[8:1:-1])
```

Output:
lroWoll



34. How Slicing Works In String Manipulation. Explain.

Syntax: Str_Object[Start_Position:End_Position:Step]

```
s = 'HelloWorld'
```

Indexing:

H	E	L	L	O	W	O	R	L	D
0	1	2	3	4	5	6	7	8	9
-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

```
print(s[-4:-2])
```

Output:
or



34. How Slicing Works In String Manipulation. Explain.

32

Syntax: Str_Object[Start_Position:End_Position:Step]

```
s = 'HelloWorld'
```

Indexing:

H	E	L	L	O	W	O	R	L	D
0	1	2	3	4	5	6	7	8	9
-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

```
print(s[::-1])
```

Output:
dlroWolleh



35. Can You Concatenate Two Tuples. If Yes, How Is It Possible? Since it is Immutable?

How To Concatenate Two Tuple:

```
t1 = (1,2,3)
t2 = (7,9,10)
t1 = t1 + t2
print("After concatenation is : ", t1)
```

Output:
After concatenation is : (1, 2, 3, 7, 9, 10)

35. Can You Concatenate Two Tuples. If Yes, How Is It Possible? Since it is Immutable?

33

Why Tuple Is Immutable and List Is Mutable?

```
tuple_1 = (1,2,3)
print(id(tuple_1)) #140180965800128
tuple_2 = (7,9,10)
print(id(tuple_2)) #140180965665600
tuple_1 = tuple_1 + tuple_2
tuple_3 = tuple_1
```

```
list_1 = [1,2,3]
print(id(list_1)) #140180965602048
list_2 = [7,9,10]
print(id(list_2)) #140180965601408
list_1.extend(list_2)
list_3 = list_1
```

35. Can You Concatenate Two Tuples. If Yes, How Is It Possible? Since it is Immutable?

Why Tuple Is Immutable and List Is Mutable?

```
tuple_1 = (1,2,3)
print(id(tuple_1)) #140180965800128
tuple_2 = (7,9,10)
print(id(tuple_2)) #140180965665600
tuple_1 = tuple_1 + tuple_2
tuple_3 = tuple_1
print("The tuple after concatenation is : ", tuple_1)
# The tuple after concatenation is : (1, 2, 3, 7, 9, 10)
print(id(tuple_3)) #140180966177280
```

Tuple Ids:
tuple_1 : #140180965800128
tuple_2 : #140180965665600
tuple_3 : #140180966177280

List Ids:
list_1 : #140180965602048
list_2 : #140180965601408
list_3 : #140180965602048

```
list_1 = [1,2,3]
print(id(list_1)) #140180965602048
list_2 = [7,9,10]
print(id(list_2)) #140180965601408
list_1.extend(list_2)
list_3 = list_1
print("The List after concatenation is : ", list_1)
# The List after concatenation is : [1, 2, 3, 7, 9, 10]
print(id(list_3)) #140180965602048
```



36. Difference Between Python Arrays and Lists

34

LIST	ARRAY
The list can store the value of different types.	It can only consist of value of same type.
The list cannot handle the direct arithmetic operations.	It can directly handle arithmetic operations.
The lists are the build-in data structure so we don't need to import it.	We need to import the array before work with the array
The lists are less compatible than the array to store the data.	An array are much compatible than the list.
It consumes a large memory.	It is a more compact in memory size comparatively list.
It is suitable for storing the longer sequence of the data item.	It is suitable for storing shorter sequence of data items.
We can print the entire list using explicit looping.	We can print the entire list without using explicit looping.

37. What Is _a, __a, __a__ in Python?

_a

- Python doesn't have real private methods, so one underline in the beginning of a variable/function/method name means it's a private variable/function/method and It is for internal use only
- We also call it weak Private

37. What Is _a, __a, __a__ in Python?

__a

- Leading double underscore tell python interpreter to rewrite name in order to avoid conflict in subclass.
- Interpreter changes variable name with class extension and that feature known as the **Mangling**.
- In Mangling python interpreter modify variable name with __.
- So Multiple time It use as the Private member because another class can not access that variable directly.
- Main purpose for __ is to use variable/method in class only If you want to use it outside of the class you can make public api.

37. What Is a, a, a in Python?

35

a

- Name with start with a and ends with a considers special methods in Python.
- Python provide this methods to use it as the operator overloading depending on the user.
- Python provides this convention to differentiate between the user defined function with the module's function

38. How To Read Multiple Values From Single Input?

By Using Split()

```
x = list(map(int, input("Enter a multiple value: ").split()))
print("List of Values: ", x)

x = [int(x) for x in input("Enter multiple value: ").split()]
print("Number of list is: ", x)

x = [int(x) for x in input("Enter multiple value: ").split(",")]
print("Number of list is: ", x)
```

KUTUM MANGOTDA

39. How To Copy and Delete A Dictionary

36

Delete By Using clear():

```
d1 = {'A':1,'B':2,'C':3}  
d1.clear()  
print(d1) #{}
```

Delete By Using pop():

```
d1 = {'A':1,'B':2,'C':3}  
print(d1) #{'A': 1, 'B': 2, 'C': 3}  
d1.pop('A')  
print(d1) #{'B': 2, 'C': 3}
```

Delete By Using del():

```
del d1['B']  
print(d1) #{'C': 3}
```



39. How To Copy and Delete A Dictionary

Copy A Dictionary Using copy():

```
d2 = {'A':1,'B':2,'C':3}  
print(d2) #{'A': 1, 'B': 2, 'C': 3}  
d3 = d2.copy()  
print(d3) #{'A': 1, 'B': 2, 'C': 3}
```

Copy A Dictionary Using '=':

```
d2 = {'A':1,'B':2,'C':3}  
print(d2) #{'A': 1, 'B': 2, 'C': 3}  
d3 = d2  
print(d3) #{'A': 1, 'B': 2, 'C': 3}
```



39. How To Copy and Delete A Dictionary

37

Copy A Dictionary Using `copy()`:

```
d2 = {'A':1,'B':2,'C':3}  
print(d2) # {'A': 1, 'B': 2, 'C': 3}  
d3 = d2.copy()  
print(d3) # {'A': 1, 'B': 2, 'C': 3}
```

Copy A Dictionary Using `'='`:

```
d2 = {'A':1,'B':2,'C':3}  
print(d2) # {'A': 1, 'B': 2, 'C': 3}  
d3 = d2  
print(d3) # {'A': 1, 'B': 2, 'C': 3}
```



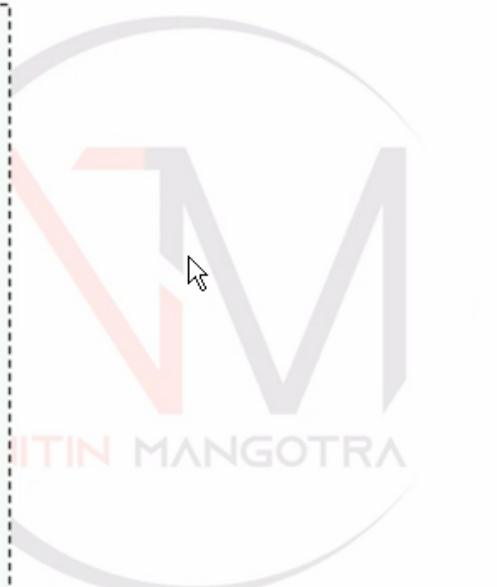
39. How To Copy and Delete A Dictionary

Benefit Of Using `copy()`:

```
d2 = {'A':1,'B':2,'C':3}  
print(d2) # {'A': 1, 'B': 2, 'C': 3}  
d3 = d2.copy()  
print(d3) # {'A': 1, 'B': 2, 'C': 3}  
del d2['B']  
print(d2) # {'A': 1, 'C': 3}  
print(d3) # {'A': 1, 'B':2, 'C': 3}
```

DrawBack Of Using `'='`

```
d2 = {'A':1,'B':2,'C':3}  
print(d2) # {'A': 1, 'B': 2, 'C': 3}  
d3 = d2  
print(d3) # {'A': 1, 'B': 2, 'C': 3}  
del d2['B']  
print(d2) # {'A': 1, 'C': 3}  
print(d3) # {'A': 1, 'C': 3}
```



Lambda function:

- ❑ It can have any number of arguments but only one expression.
- ❑ The expression is evaluated and returned.
- ❑ Lambda functions can be used wherever function objects are required.

Anonymous function:

- ❑ In Python, **Anonymous function** is a function that is defined without a name.
- ❑ While normal functions are defined using the `def` keyword, Anonymous functions are defined using the `lambda` keyword.
- ❑ Hence, anonymous functions are also called lambda functions.

40. Difference Between Anonymous and Lambda Function

Syntax:

```
lambda [arguments] : expression
```

Example:

```
square = lambda x : x * x
square(5) #25
```

The above lambda function definition is the same as the following function:

```
def square(x):
    return x * x
```



Anonymous Function: We can declare a lambda function and call it as an anonymous function, without assigning it to a variable.

```
print((lambda x: x*x)(5))
```

Above, `lambda x: x*x` defines an anonymous function and call it once by passing arguments in the parenthesis `(lambda x: x*x)(5)`.

41. How to achieve Multiprocessing and Multithreading in Python?

39

Multithreading:

- ❑ It is a technique where multiple threads are spawned by a process to do different tasks, at about the same time, just one after the other.
- ❑ This gives you the illusion that the threads are running in parallel, but they are actually run in a concurrent manner.
- ❑ In Python, the Global Interpreter Lock (GIL) prevents the threads from running simultaneously.

41. How to achieve Multiprocessing and Multithreading in Python?

Multiprocessing:

- ❑ It is a technique where parallelism in its truest form is achieved.
- ❑ Multiple processes are run across multiple CPU cores, which do not share the resources among them.
- ❑ Each process can have many threads running in its own memory space.
- ❑ In Python, each process has its own instance of Python interpreter doing the job of executing the instructions.

41. How to achieve Multiprocessing and Multithreading in Python?

```
# importing the multiprocessing module
import multiprocessing

def print_cube(num):
    print("Cube: {}".format(num * num * num))

def print_square(num):
    print("Square: {}".format(num * num))

if __name__ == "__main__":
    # creating processes
    p1 = multiprocessing.Process(target=print_square, args=(10,))
    p2 = multiprocessing.Process(target=print_cube, args=(10,))
    p1.start()
    p2.start()
    # wait until process 1 is finished
    p1.join()
    p2.join()

    # both processes finished
    print("Done!")
```

```
# A multithreaded program in python
import time
from threading import Thread
num= 0

# The bottleneck of the code which is CPU-bound
def upgrade(n):
    while num<400000000:
        num=num+1

# Creation of multiple threads
t1 = Thread(target=upgrade, args=(num//2,))
t2 = Thread(target=upgrade, args=(num//2,))

# multi thread architecture, recording time
start = time.time()
t1.start()
t2.start()
t1.join()
t2.join()
end = time.time()

print('Time taken in seconds -', end - start)
```

42. What is GIL. Explain

40

- ❑ The Global Interpreter Lock (GIL) of Python allows only one thread to be executed at a time. It is often a hurdle, as it does not allow multi-threading in Python to save time.
- ❑ The Python Global Interpreter Lock or GIL, in simple words, is a mutex (or a lock) that allows only one thread to hold the control of the Python interpreter.
- ❑ This means that only one thread can be in a state of execution at any point in time. The impact of the GIL isn't visible to developers who execute single-threaded programs, but it can be a performance bottleneck in CPU-bound and multi-threaded code.
- ❑ Since the GIL allows only one thread to execute at a time even in a multi-threaded architecture with more than one CPU core, the GIL has gained a reputation as an "infamous" feature of Python.
- ❑ Basically, GIL in Python doesn't allow multi-threading which can sometimes be considered as a disadvantage.

43. How Class and Object Created in Python?

- ❑ Python is an object-oriented programming language.
- ❑ Almost everything in Python is an object, with its properties and methods.
- ❑ A Class is like an object constructor, or a "blueprint" for creating objects.

Create a Class:

To create a class, use the keyword 'class':

class MyClass:

 x = 5

Create Object:

Now we can use the class named MyClass to create objects:
(Create an object named obj, and print the value of x:)

```
obj= MyClass()  
print(obj.x)
```

Namespace:

- ❑ In python we deal with variables, functions, libraries and modules etc.
- ❑ There is a chance the name of the variable you are going to use is already existing as name of another variable or as the name of another function or another method.
- ❑ In such scenario, we need to learn about how all these names are managed by a python program. This is the concept of **namespace**.

44. Explain Namespace and Its Types in Python.

Categories Of Namespace: Following are the three categories of namespace

- ❑ **Local Namespace:** All the names of the functions and variables declared by a program are held in this namespace. This namespace exists as long as the program runs.
- ❑ **Global Namespace:** This namespace holds all the names of functions and other variables that are included in the modules being used in the python program. It includes all the names that are part of the Local namespace.
- ❑ **Built-in Namespace:** This is the highest level of namespace which is available with default names available as part of the python interpreter that is loaded as the programming environment. It includes Global Namespace which in turn include the local namespace.

We can access all the names defined in the built-in namespace as follows.

```
builtin_names = dir(__builtins__)
for name in builtin_names:
    print(name)
```



45. Explain Recursion by Reversing a List.

42

```
def reverseList(lst):
    if not lst:
        return []
    return [lst[-1]] + reverseList(lst[:-1])

print(reverseList([1, 2, 3, 4, 5]))
```

Output:
[5,4,3,2,1]

NITIN MANGOTRA

46. What are Unittests in Python

Unit Testing is the first level of software testing where the smallest testable parts of a software are tested. This is used to validate that each unit of the software performs as designed. The unittest test framework is python's xUnit style framework. This is how you can import it.

```
import unittest

- Unit testing is a software testing method by which individual units of source code are put under various tests to determine whether they are fit for use (Source). It determines and ascertains the quality of your code.
- Generally, when the development process is complete, the developer codes criteria, or the results that are known to be potentially practical and useful, into the test script to verify a particular unit's correctness. During test case execution, various frameworks log tests that fail any criterion and report them in a summary.
- The developers are expected to write automated test scripts, which ensures that each and every section or a unit meets its design and behaves as expected.
- Though writing manual tests for your code is definitely a tedious and time-consuming task, Python's built-in unit testing framework has made life a lot easier.
- The unit test framework in Python is called unittest, which comes packaged with Python.
- Unit testing makes your code future proof since you anticipate the cases where your code could potentially fail or produce a bug. Though you cannot predict all of the cases, you still address most of them.
```

47. How to use Map, Filter and Reduce Function in Python?

43

Map() Function:

The map() function iterates through all items in the given iterable and executes the function we passed as an argument on each of them.

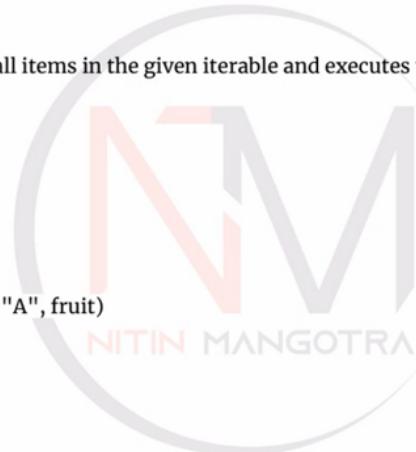
The syntax is:

```
map(function, iterable(s))
```

```
fruit = ["Apple", "Banana", "Pear"]
map_object = map(lambda s: s[0] == "A", fruit)
print(list(map_object))
```

Output:

```
[True, False, False]
```



47. How to use Map, Filter and Reduce Function in Python?

Filter() Function

The filter() function takes a function object and an iterable and creates a new list. As the name suggests, filter() forms a new list that contains only elements that satisfy a certain condition, i.e. the function we passed returns True.

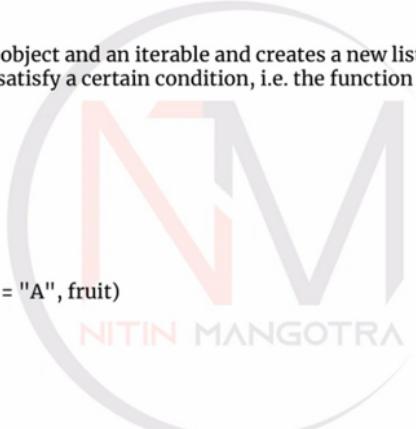
The syntax is:

```
filter(function, iterable(s))
```

```
fruit = ["Apple", "Banana", "Pear"]
filter_object = filter(lambda s: s[0] == "A", fruit)
print(list(filter_object))
```

Output:

```
['Apple', 'Apricot']
```



47. How to use Map, Filter and Reduce Function in Python?

44

Reduce() Function

The reduce() Function works differently than map() and filter(). It does not return a new list based on the function and iterable we've passed. Instead, it returns a single value. Also, in Python 3 reduce() isn't a built-in function anymore, and it can be found in the functools module.

The syntax is:

```
reduce(function, sequence[, initial])
```

```
from functools import reduce
list = [2, 4, 7, 3]
print(reduce(lambda x, y: x + y, list))
print("With an initial value: " + str(reduce(lambda x, y: x + y, list, 10)))
```

Output:

16

With an initial value: 26



48. Difference Between Shallow Copy and Deep Copy

Shallow Copy:

Shallow copies duplicate as little as possible. A shallow copy of a collection is a copy of the collection structure, not the elements. With a shallow copy, two collections now share the individual elements.

Shallow copying is creating a new object and then copying the non static fields of the current object to the new object. If the field is a value type, a bit by bit copy of the field is performed. If the field is a reference type, the reference is copied but the referred object is not, therefore the original object and its clone refer to the same object.

Deep Copy:

Deep copies duplicate everything. A deep copy of a collection is two collections with all of the elements in the original collection duplicated.

Deep copy is creating a new object and then copying the non-static fields of the current object to the new object. If a field is a value type, a bit by bit copy of the field is performed. If a field is a reference type, a new copy of the referred object is performed. A deep copy of an object is a new object with entirely new instance variables, it does not share objects with the old. While performing Deep Copy the classes to be cloned must be flagged as [Serializable].

*How an object
is copied?*

50. What does term MONKEY PATCHING refer to in python?

45

In Python, the term monkey patch refers to dynamic (or run-time) modifications of a class or module. In Python, we can actually change the behavior of code at run-time.

```
# monkey.py
class A:
    def func(self):
        print ("func() is called")
```

We use above module (monkey) in below code and change behavior of func() at run-time by assigning different value.

```
import monkey
def monkey_func(self):
    print ("monkey_func() is called")

# replacing address of "func" with "monkey_func"
monkey.A.func = monkey_func
obj = monkey.A()

# calling function "func" whose address got replaced
# with function "monkey_func()"
obj.func()
```

Examples:

Output :monkey_func() is called



51. What is Operator Overloading & Dunder Method.

- ❑ Dunder methods in Python are special methods.
- ❑ In Python, we sometimes see method names with a double underscore (____), such as the `__init__` method that every class has. These methods are called “dunder” methods.
- ❑ In Python, Dunder methods are used for operator overloading and customizing some other function’s behavior.



Some Examples:

<code>+</code>	<code>__add__</code> (<code>self, other</code>)
<code>-</code>	<code>__sub__</code> (<code>self, other</code>)
<code>*</code>	<code>__mul__</code> (<code>self, other</code>)
<code>/</code>	<code>__truediv__</code> (<code>self, other</code>)
<code>//</code>	<code>__floordiv__</code> (<code>self, other</code>)
<code>%</code>	<code>__mod__</code> (<code>self, other</code>)
<code>**</code>	<code>__pow__</code> (<code>self, other</code>)
<code>>></code>	<code>__rshift__</code> (<code>self, other</code>)
<code><<</code>	<code>__lshift__</code> (<code>self, other</code>)
<code>&</code>	<code>__and__</code> (<code>self, other</code>)
<code> </code>	<code>__or__</code> (<code>self, other</code>)
<code>^</code>	<code>__xor__</code> (<code>self, other</code>)

52. Draw Pattern.

```
# This is the example of print simple pyramid pattern
n = int(input("Enter the number of rows"))
# outer loop to handle number of rows
for i in range(0, n):
    # inner loop to handle number of columns
    # values is changing according to outer loop
    for j in range(0, i + 1):
        # printing stars
        print("* ", end="")
    # ending line after each row
    print()
```

Output:

*
* *
* * *
* * * *
* * * * *

1
2 2
3 3 3
4 4 4 4
5 5 5 5 5



