History of Python

- Created in 1990 by Guido Van Rossum
- Python 3 released in 2008

Name	Type	Description
Integers	int	Whole numbers, such as: 3 300 200
Floating point	float	Numbers with a decimal point: 2.3 4.6 100.0
Strings	str	Ordered sequence of characters: "hello" 'Sammy' "2000" "楽しい" Strings are immutable, we cannot change it. We can reassign it.
Lists	list	Ordered sequence of objects: [10,"hello",200.3] Lists are mutable
Dictionaries	dict	Unordered Key:Value pairs: {"mykey" : "value" , "name" : "Frankie"}
Tuples	tup	Ordered immutable sequence of objects: (10,"hello",200.3) Tuples are immutable
Sets	set	Unordered collection of unique objects: {"a", "b"}
Booleans	bool	Logical value indicating True or False

Python is a high-level, interpreted, general-purpose programming language. Being a general-purpose language, it can be used to build almost any type of application with the right tools/libraries.

Python is Dynamic Typing language. No need to define variable type explicitly.

- Pros of Dynamic Typing:
  - Very easy to work with
  - Faster development time
- Cons of Dynamic Typing:
  - May result in bugs for unexpected data types!

You need to be aware of type()

Slicing

- Slicing allows you to grab a subsection of multiple characters, a “slice” of the string.
- This has the following syntax:
  - [start:stop:step]
- start is a numerical index for the slice start
- stop is the index you will go up to (but not include)

step is the size of the “jump” you take
Reverse a string: *msg = ‘abc’ msg[::-1] # output: ‘cba’*

Q5. What is Scope in Python?

Every object in Python functions within a scope. A scope is a block of code where an object in Python remains relevant. Namespaces uniquely identify all the objects inside a program. However, these namespaces also have a scope defined for them where you could use their objects without any prefix. A few examples of scope created during code execution in Python are as follows:

- A **local scope** refers to the local objects available in the current function.
- A **global scope** refers to the objects available throughout the code execution since their inception.
- A **module-level scope** refers to the global objects of the current module accessible in the program.
- An **outermost scope** refers to all the built-in names callable in the program. The objects in this scope are searched last to find the name referenced.

Note: Local scope objects can be synced with global scope objects using keywords such as **global**.

Q6. What are lists and tuples? What is the key difference between the two?

Lists and **Tuples** are both s**equence data types** that can store a collection of objects in Python. The objects stored in both sequences can have **different data types**. Lists are represented with **square brackets** ['sara', 6, 0.19], while tuples are represented with **parantheses** ('ansh', 5, 0.97).

The key difference between the two is that while **lists are mutable**, **tuples** on the other hand are **immutable** objects. This means that lists can be modified, appended or sliced on the go but tuples remain constant and cannot be modified in any manner.	my_tuple = ('sara', 6, 5, 0.97)
	my_list = ['sara', 6, 5, 0.97]
	print(my_tuple[0]) # output => 'sara'
	print(my_list[0]) # output => 'sara'
	my_tuple[0] = 'ansh' # modifying tuple => throws an error
	my_list[0] = 'ansh' # modifying list => list modified
	print(my_tuple[0]) # output => 'sara'
	
	print(my_list[0]) # output => 'ansh'

8. What is pass in Python?

The pass keyword represents a null operation in Python. It is generally used for the purpose of filling up empty blocks of code which may execute during runtime but has yet to be written.

9. What are modules and packages in Python?

Python packages and Python modules are two mechanisms that allow for **modular programming** in Python. Modularizing has several advantages -

- **Simplicity**: Working on a single module helps you focus on a relatively small portion of the problem at hand. This makes development easier and less error-prone.
- **Maintainability**: Modules are designed to enforce logical boundaries between different problem domains. If they are written in a manner that reduces interdependency, it is less likely that modifications in a module might impact other parts of the program.
- **Reusability**: Functions defined in a module can be easily reused by other parts of the application.
- **Scoping**: Modules typically define a separate namespace, which helps avoid confusion between identifiers from other parts of the program.

Modules, in general, are simply Python files with a .py extension and can have a set of functions, classes, or variables defined and implemented. They can be imported and initialized once using the import statement. If partial functionality is needed, import the requisite classes or functions using from foo import bar.

Packages allow for hierarchial structuring of the module namespace using **dot notation**.
As, **modules** help avoid clashes between global variable names, in a similar manner, **packages** help avoid clashes between module names.
Creating a package is easy since it makes use of the system's inherent file structure. So just stuff the modules into a folder and there you have it, the folder name as the package name. Importing a module or its contents from this package requires the package name as prefix to the module name joined by a dot.

10. What are global, protected and private attributes in Python?

- **Global** variables are public variables that are defined in the global scope. To use the variable in the global scope inside a function, we use the global keyword.
- **Protected** attributes are attributes defined with an underscore prefixed to their identifier eg. _sara. They can still be accessed and modified from outside the class they are defined in but a responsible developer should refrain from doing so.
- **Private** attributes are attributes with double underscore prefixed to their identifier eg. __ansh. They cannot be accessed or modified from the outside directly and will result in an AttributeError if such an attempt is made.

11. What is the use of self in Python?

Self is used to represent the instance of the class. With this keyword, you can access the attributes and methods of the class in python. It binds the attributes with the given arguments. self is used in different places and often thought to be a keyword. But unlike in C++, self is not a keyword in Python.

12. What is __init__?

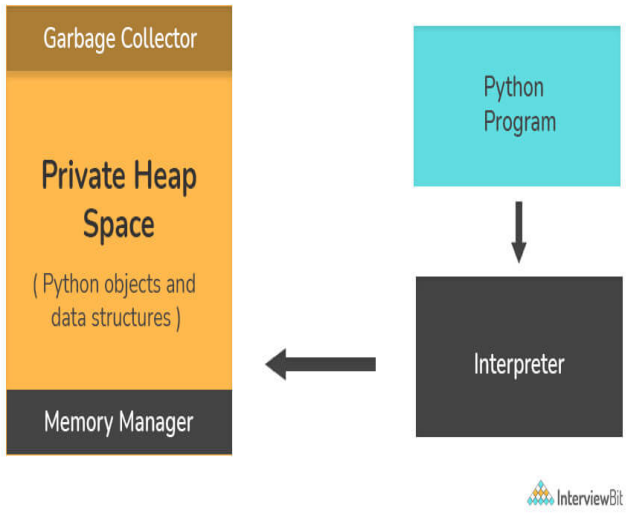
__init__ is a contructor method in Python and is automatically called to allocate memory when a new object/instance is created. All classes have a **__init__** method associated with them. It helps in distinguishing methods and attributes of a class from local variables.

18. What is the difference between Python Arrays and lists?

- Arrays in python can only contain elements of same data types i.e., data type of array should be homogeneous. It is a thin wrapper around C language arrays and consumes far less memory than lists.
- Lists in python can contain elements of different data types i.e., data type of lists can be heterogeneous. It has the disadvantage of consuming large memory.

19. How is memory managed in Python?

- Memory management in Python is handled by the **Python Memory Manager**. The memory allocated by the manager is in form of a **private heap space** dedicated to Python. All Python objects are stored in this heap and being private, it is inaccessible to the programmer. Though, python does provide some core API functions to work upon the private heap space.



- Additionally, Python has an in-built garbage collection to recycle the unused memory for the private heap space.

20. What are Python namespaces? Why are they used?

A namespace in Python ensures that object names in a program are unique and can be used without any conflict. Python implements these namespaces as dictionaries with 'name as key' mapped to a corresponding 'object as value'. This allows for multiple namespaces to use the same name and map it to a separate object. A few examples of namespaces are as follows:

- **Local Namespace** includes local names inside a function. the namespace is temporarily created for a function call and gets cleared when the function returns.
- **Global Namespace** includes names from various imported packages/ modules that are being used in the current project. This namespace is created when the package is imported in the script and lasts until the execution of the script.
- **Built-in Namespace** includes built-in functions of core Python and built-in names for various types of exceptions.

22. What are decorators in Python?

Decorators in Python are essentially functions that add functionality to an existing function in Python without changing the structure of the function itself. They are represented the `@decorator_name` in Python and are called in a bottom-up fashion. For example:

```
# decorator function to convert to lowercase
def lowercase_decorator(function):
  def wrapper():
    func = function()
    string_lowercase = func.lower()
    return string_lowercase

  return wrapper
```

```
@lowercase_decorator # this is executed first
def hello():
  return 'Hello World'

hello()  # output => [ 'hello world' ]
```

The beauty of the decorators lies in the fact that besides adding functionality to the output of the method, they can even **accept arguments** for functions and can further modify those arguments before passing it to the function itself. The **inner nested function**, i.e. 'wrapper' function, plays a significant role here. It is implemented to enforce **encapsulation** and thus, keep itself hidden from the global scope.

```
# decorator function to capitalize names
def names_decorator(function):
  def wrapper(arg1, arg2):
    arg1 = arg1.capitalize()
    arg2 = arg2.capitalize()
    string_hello = function(arg1, arg2)
    return string_hello

  return wrapper
```

```
@names_decorator
def say_hello(name1, name2):
  return 'Hello ' + name1 + '! Hello ' + name2 + '!'

say_hello('sara', 'ansh')  # output => 'Hello Sara! Hello Ansh!'
```

24. What is lambda in Python? Why is it used?

Lambda is an anonymous function in Python, that can accept any number of arguments, but can only have a single expression. It is generally used in situations requiring an anonymous function for a short time period. Lambda functions can be used in either of the two ways:

Assigning lambda functions to a variable:

```
mul = lambda a, b : a * b

print(mul(2, 5))    # output => 10
```

Wrapping lambda functions inside another function:

```
def myWrapper(n):
 return lambda a : a * n
mulFive = myWrapper(5)
print(mulFive(2))    # output => 10
```

25. How do you copy an object in Python?

In Python, the assignment statement (= operator) does not copy objects. Instead, it creates a binding between the existing object and the target variable name. To create copies of an object in Python, we need to use the **copy** module. Moreover, there are two ways of creating copies for the given object using the **copy** module -

Shallow Copy is a bit-wise copy of an object. The copied object created has an exact copy of the values in the original object. If either of the values is a reference to other objects, just the reference addresses for the same are copied.
Deep Copy copies all values recursively from source to target object, i.e. it even duplicates the objects referenced by the source object.

```
from copy import copy, deepcopy
list_1 = [1, 2, [3, 5], 4]

## shallow copy
list_2 = copy(list_1)
list_2[3] = 7
list_2[2].append(6)
list_2    # output => [1, 2, [3, 5, 6], 7]
list_1    # output => [1, 2, [3, 5, 6], 4]
```

```
## deep copy
list_3 = deepcopy(list_1)
list_3[3] = 8
list_3[2].append(7)
list_3    # output => [1, 2, [3, 5, 6, 7], 8]
list_1    # output => [1, 2, [3, 5, 6], 4]
```

27. What is pickling and unpickling?

Python library offers a feature - **serialization** out of the box. Serializing an object refers to transforming it into a format that can be stored, so as to be able to deserialize it, later on, to obtain the original object. Here, the **pickle** module comes into play.

Pickling:

- Pickling is the name of the serialization process in Python. Any object in Python can be serialized into a byte stream and dumped as a file in the memory. The process of pickling is compact but pickle objects can be compressed further. Moreover, pickle keeps track of the objects it has serialized and the serialization is portable across versions.
- The function used for the above process is pickle.dump().

Unpickling:

- Unpickling is the complete inverse of pickling. It deserializes the byte stream to recreate the objects stored in the file and loads the object to memory.
- The function used for the above process is pickle.load().

28. What are generators in Python?

Generators are functions that return an iterable collection of items, one at a time, in a set manner. Generators, in general, are used to create iterators with a different approach. They employ the use of yield keyword rather than return to return a **generator** object.
Let's try and build a generator for fibonacci numbers -

```
## generate fibonacci numbers upto n
def fib(n):
    p, q = 0, 1
    while(p < n):
        yield p
        p, q = q, p + q
x = fib(10)    # create generator object
```

```
## iterating using __next__(), for Python2, use next()
x.__next__()    # output => 0
x.__next__()    # output => 1
x.__next__()    # output => 1
x.__next__()    # output => 2
x.__next__()    # output => 3
x.__next__()    # output => 5
x.__next__()    # output => 8
x.__next__()    # error

## iterating using loop
for i in fib(10):

    print(i)    # output => 0 1 1 2 3 5 8
```

30. What is the use of help() and dir() functions?

help() function in Python is used to display the documentation of modules, classes, functions, keywords, etc. If no parameter is passed to the help() function, then an interactive **help utility** is launched on the console.
dir() function tries to return a valid list of attributes and methods of the object it is called upon. It behaves differently with different objects, as it aims to produce the most relevant data, rather than the complete information.

- For Modules/Library objects, it returns a list of all attributes, contained in that module.
- For Class Objects, it returns a list of all valid attributes and base attributes.
- With no arguments passed, it returns a list of attributes in the current scope.

31. What is the difference between .py and .pyc files?

- .py files contain the source code of a program. Whereas, .pyc file contains the bytecode of your program. We get bytecode after compilation of .py file (source code). .pyc files are not created for all the files that you run. It is only created for the files that you import.
- Before executing a python program python interpreter checks for the compiled files. If the file is present, the virtual machine executes it. If not found, it checks for .py file. If found, compiles it to .pyc file and then python virtual machine executes it.
- Having .pyc file saves you the compilation time.

33. How are arguments passed by value or by reference in python?

- **Pass by value**: Copy of the actual object is passed. Changing the value of the copy of the object will not change the value of the original object.
- **Pass by reference**: Reference to the actual object is passed. Changing the value of the new object will change the value of the original object.

```
def appendNumber(arr):
    arr.append(4)
arr = [1, 2, 3]
print(arr)  #Output: => [1, 2, 3]
appendNumber(arr)
print(arr)  #Output: => [1, 2, 3, 4]
```

In Python, arguments are passed by reference, i.e., reference to the actual object is passed.

36. Explain split() and join() functions in Python?

You can use **split()** function to split a string based on a delimiter to a list of strings.
 	
You can use **join()** function to join a list of strings based on a delimiter to give a single string.
	string = "This is a string."
	#delimiter is 'space' character or ' '
	string_list = string.split(' ')
	
	print(string_list) #output: ['This', 'is', 'a', 'string.']
	print(' '.join(string_list)) #output: This is a string.

37. What does *args and **kwargs mean?

***args**

- *args is a special syntax used in the function definition to pass variable-length arguments.
- "*" means variable length and "args" is the name used by convention. You can use any other.
- args is a Tuple

****kwargs**

- **kwargs is a special syntax used in the function definition to pass variable-length keyworded arguments.
- Here, also, "kwargs" is used just by convention. You can use any other name.
- Keyworded argument means a variable that has a name when passed to a function.
- It is actually a dictionary of the variable names and its value.

*args	**kwargs

def multiply(a, b, *argv):
mul = a * b
for num in argv:
mul *= num
return mul
print(multiply(1, 2, 3, 4, 5)) #output:
120
	def tellArguments(**kwargs):
	for key, value in kwargs.items():
	print(key + ": " + value)
	tellArguments(arg1 = "argument 1", arg2 = "argument 2", arg3 = "argument 3")
	#output:
	# arg1: argument 1
	# arg2: argument 2
	# arg3: argument 3

Inheritance

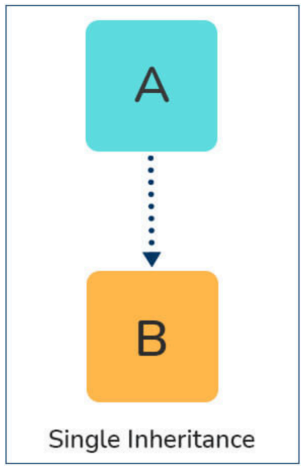
Inheritance gives the power to a class to access all attributes and methods of another class.

Single Inheritance: Child class derives members of one parent class.

```
# Parent class
class ParentClass:
    def par_func(self):
        print("I am parent class function")

# Child class
class ChildClass(ParentClass):
    def child_func(self):
        print("I am child class function")

# Driver code
obj1 = ChildClass()
obj1.par_func()
obj1.child_func()
```



Multi-level Inheritance: The members of the parent class, A, are inherited by child class which is then inherited by another child class, B. The features of the base class and the derived class are further inherited into the new derived class, C. Here, A is the grandfather class of class C.

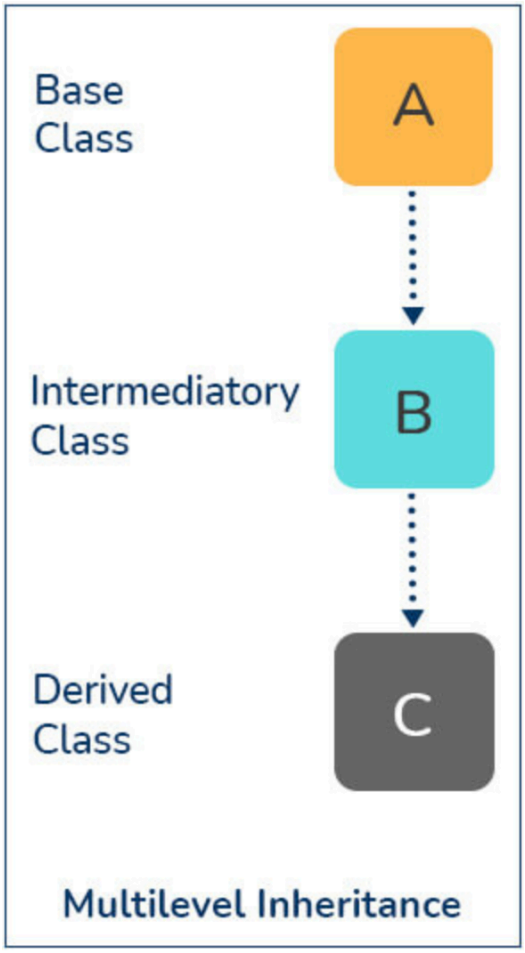
```
# Parent class
class A:
  def __init__(self, a_name):
    self.a_name = a_name

# Intermediate class
class B(A):
  def __init__(self, b_name, a_name):
    self.b_name = b_name
    # invoke constructor of class A
    A.__init__(self, a_name)

# Child class
class C(B):
  def __init__(self,c_name, b_name, a_name):
    self.c_name = c_name
    # invoke constructor of class B
    B.__init__(self, b_name, a_name)

  def display_names(self):
    print("A name : ", self.a_name)
    print("B name : ", self.b_name)
    print("C name : ", self.c_name)

#  Driver code
obj1 = C('child', 'intermediate', 'parent')
print(obj1.a_name)
obj1.display_names()
```



Multiple Inheritance: This is achieved when one child class derives members from more than one parent class. All features of parent classes are inherited in the child class.

```
# Parent class1
class Parent1:
  def parent1_func(self):
    print("Hi I am first Parent")

# Parent class2
class Parent2:
  def parent2_func(self):
    print("Hi I am second Parent")

# Child class
class Child(Parent1, Parent2):
  def child_func(self):
    self.parent1_func()
    self.parent2_func()

# Driver's code
obj1 = Child()
obj1.child_func()
```