Python Interview Questions for Freshers

1. What is Python? What are the benefits of using Python

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. Additionally, python supports objects, modules, threads, exception-handling, and automatic memory management which help in modelling real-world problems and building applications to solve these problems.

Benefits of using Python:

Python is a general-purpose programming language that has a simple, easy-to-learn syntax that emphasizes readability and therefore reduces the cost of program maintenance. Moreover, the language is capable of scripting, is completely open-source, and supports third-party packages encouraging modularity and code reuse.

Its high-level data structures, combined with dynamic typing and dynamic binding, attract a huge community of developers for Rapid Application Development and deployment.

2. What is a dynamically typed language?

Before we understand a dynamically typed language, we should learn about what typing is. Typing refers to type-checking in programming languages. In a strongly-typed language, such as Python, "1" + 2 will result in a type error since these languages don't allow for "type-coercion" (implicit conversion of data types). On the other hand, a weakly-typed language, such as Javascript, will simply output "12" as result.

Type-checking can be done at two stages -

Static - Data Types are checked before execution.

Dynamic - Data Types are checked during execution.

Python is an interpreted language, executes each statement line by line and thus type-checking is done on the fly, during execution. Hence, Python is a Dynamically Typed Language.

3. What is an Interpreted language?

An Interpreted language executes its statements line by line. Languages such as Python, Javascript, R, PHP, and Ruby are prime examples of Interpreted languages. Programs written in an interpreted language runs directly from the source code, with no intermediary compilation step.

4. What is PEP 8 and why is it important?

PEP stands for **Python Enhancement Proposal**. A PEP is an official design document providing information to the Python community or describing a new feature for Python or its processes. **PEP 8** is especially important since it documents the style guidelines for Python Code. Apparently contributing to the Python open-source community requires you to follow these style guidelines sincerely and strictly.

5. 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**.

6. 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). But what is the real difference between the two? 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. You can run the following example on Python IDLE to confirm the difference:

```
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'
```

7. What are the common built-in data types in Python?

There are several built-in data types in Python. Although, Python doesn't require data types to be defined explicitly during variable declarations type errors are likely to occur if the knowledge of data types and their compatibility with each other are neglected. Python provides type() and isinstance() functions to check the type of these variables. These data types can be grouped into the following categories-

- **None Type:**
  None keyword represents the null values in Python. Boolean equality operation can be performed using these NoneType objects.

Class Name	Description
NoneType	Represents the **NULL** values in Python.

- **Numeric Types:**
  There are three distinct numeric types - **integers, floating-point numbers**, and **complex numbers**. Additionally, **booleans** are a sub-type of integers.

Class Name	Description
int	Stores integer literals including hex, octal and binary numbers as integers
float	Stores literals containing decimal values and/or exponent signs as floating-point numbers
complex	Stores complex numbers in the form (A + Bj) and has attributes: real and imag
bool	Stores boolean value (True or False).

Note: *The standard library also includes* ***fractions*** *to store rational numbers and* ***decimal*** *to store floating-point numbers with user-defined precision.*

- **Sequence Types:**
  According to Python Docs, there are three basic Sequence Types - **lists, tuples,** and **range** objects. Sequence types have the in and not in operators defined for their traversing their elements. These operators share the same priority as the comparison operations.

Class Name	Description
list	Mutable sequence used to store collection of items.
tuple	Immutable sequence used to store collection of items.
range	Represents an immutable sequence of numbers generated during execution.
str	Immutable sequence of Unicode code points to store textual data.

Note: The standard library also includes additional types for processing:
1. **Binary data** such as bytearray bytes memoryview , and
2. **Text strings** such as str.

- **Mapping Types:**

A mapping object can map hashable values to random objects in Python. Mappings objects are mutable and there is currently only one standard mapping type, the ***dictionary***.

Class Name	Description
dict	Stores comma-separated list of **key: value** pairs

- **Set Types:**
  Currently, Python has two built-in set types
  - set and **frozenset**. **set** type is mutable and supports methods like add() and remove(). **frozenset** type is immutable and can't be modified after creation.

Class Name	Description
set	Mutable unordered collection of distinct hashable objects.
frozenset	Immutable collection of distinct hashable objects.

Note: *set is mutable and thus cannot be used as key for a dictionary. On the other hand, frozenset is immutable and thus, hashable, and can be used as a dictionary key or as an element of another set.*

- **Modules:**
  Module is an additional built-in type supported by the Python Interpreter. It supports one special operation, i.e., **attribute access**: mymod.myobj, where mymod is a module and **myobj** references a name defined in m's symbol table. The module's symbol table resides in a very special attribute of the module **__dict__**, but direct assignment to this module is neither possible nor recommended.
- **Callable Types:**
  Callable types are the types to which function call can be applied. They can be **user-defined functions, instance methods, generator functions**, and some other **built-in functions, methods** and **classes**. Refer to the documentation at docs.python.org for a detailed view of the **callable types**.

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. Without the pass statement in the following code, we may run into some errors during code execution.

```
def myEmptyFunc():

  # do nothing

  pass

myEmptyFunc()    # nothing happens

## Without the pass keyword

# File "<stdin>", line 3

# IndentationError: expected an indented block
```

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.

class definition

```
class Student:

    def __init__(self, fname, lname, age, section):

        self.firstname = fname

        self.lastname = lname

        self.age = age

        self.section = section

# creating a new object

stu1 = Student("Sara", "Ansh", 22, "A2")
```

13. What is break, continue and pass in Python?

Break : The break statement terminates the loop immediately and the control flows to the statement after the body of the loop.

Continue : The continue statement terminates the current iteration of the statement, skips the rest of the code in the current iteration and the control flows to the next iteration of the loop.

Pass : As explained above, the pass keyword in Python is generally used to fill up empty blocks and is similar to an empty statement represented by a semi-colon in languages such as Java, C++, Javascript, etc.

```
pat = [1, 3, 2, 1, 2, 3, 1, 0, 1, 3]
for p in pat:
    pass
    if (p == 0): pat = [1, 3, 2, 1, 2, 3, 1, 0, 1, 3]
for p in pat:
    pass
    if (p == 0):
        current = p
        break
    elif (p % 2 == 0):
        continue
```

```
    print(p)    # output => 1 3 1 3 1
print(current)    # output => 0
        current = p

        break
    elif (p % 2 == 0):
        continue
    print(p)    # output => 1 3 1 3 1

print(current)    # output => 0
```

14. What are unit tests in Python?

- Unit test is a unit testing framework of Python.
- Unit testing means testing different components of software separately. Can you think about why unit testing is important? Imagine a scenario, you are building software that uses three components namely A, B, and C. Now, suppose your software breaks at a point time. How will you find which component was responsible for breaking the software? Maybe it was component A that failed, which in turn failed component B, and this actually failed the software. There can be many such combinations.
- This is why it is necessary to test each and every component properly so that we know which component might be highly responsible for the failure of the software.

15. What is docstring in Python?

- Documentation string or docstring is a multiline string used to document a specific code segment.
- The docstring should describe what the function or method does.

16. What is slicing in Python?

- As the name suggests, ‘slicing’ is taking parts of.
- Syntax for slicing is **[start : stop : step]**
- **start** is the starting index from where to slice a list or tuple
- **stop** is the ending index or where to sop.
- **step** is the number of steps to jump.
- Default value for **start** is 0, **stop** is number of items, **step** is 1.
- Slicing can be done on **strings, arrays, lists**, and **tuples**.

Example:

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(numbers[1 : : 2])  #output : [2, 4, 6, 8, 10]
```

17. Explain how can you make a Python Script executable on Unix?

- Script file must begin with **#!/usr/bin/env python**

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

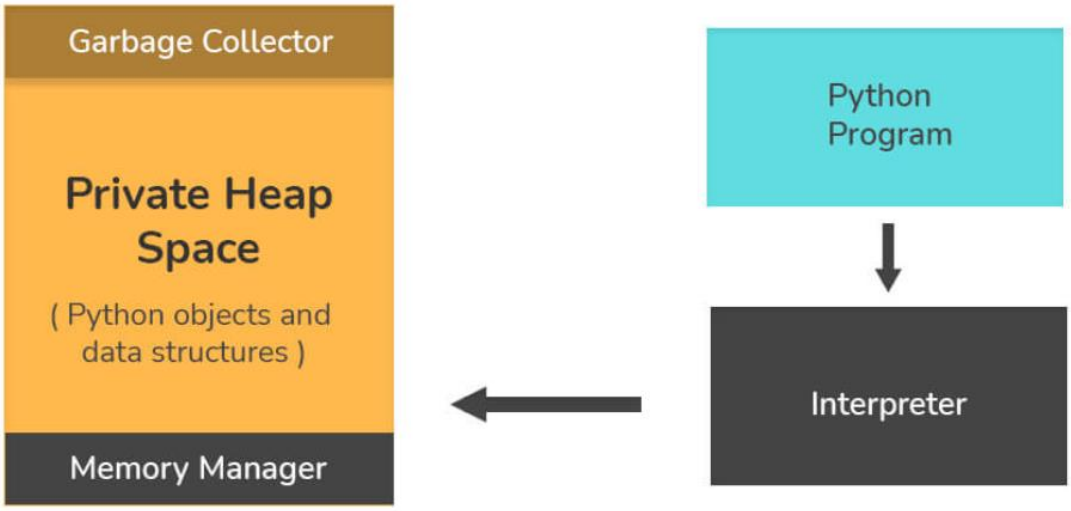
- Arrays in python can only contain elements of same data types i.e., data type of array should be homogeneous. It is a thin wrapper around C language arrays and consumes far less memory than lists.
- Lists in python can contain elements of different data types i.e., data type of lists can be heterogeneous.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

```
 import array
a = array.array('i', [1, 2, 3])
for i in a:
    print(i, end=' ')    #OUTPUT: 1 2 3
a = array.array('i', [1, 2, 'string'])    #OUTPUT: TypeError: an integer is required (got type str)
a = [1, 2, 'string']
for i in a:
        print(i, end=' ')    #OUTPUT: 1 2 string
```

Python Interview Questions for Experienced

1. 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.

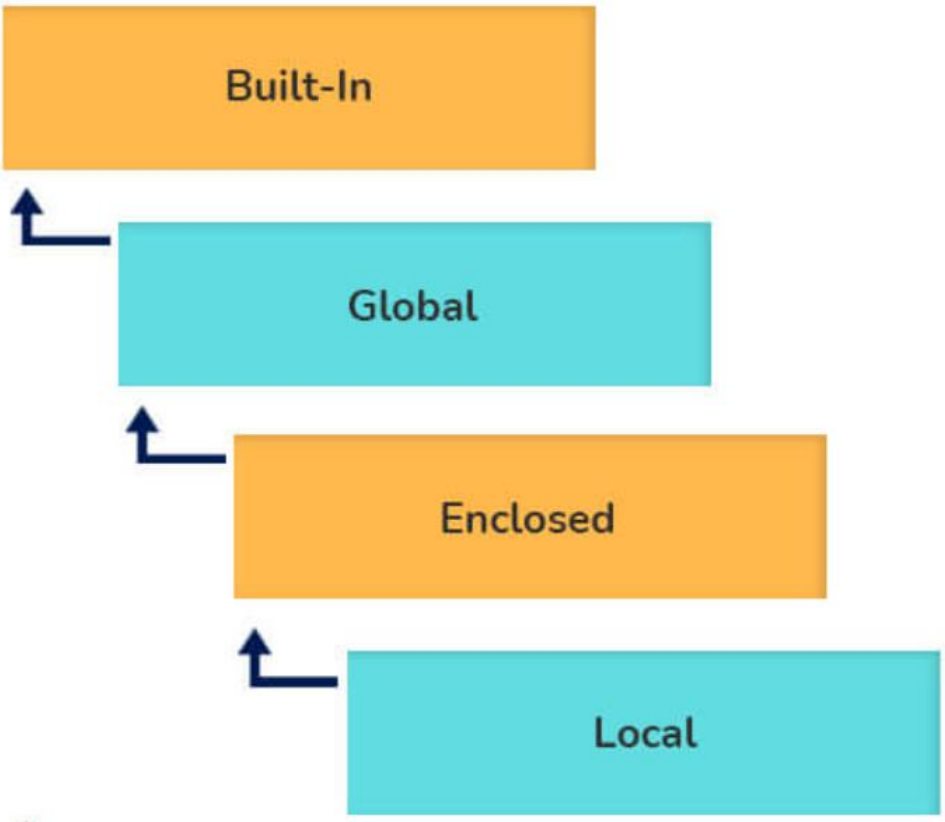


2. What are Python namespaces? Why are they used?

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.

The **lifecycle of a namespace** depends upon the scope of objects they are mapped to. If the scope of an object ends, the lifecycle of that namespace comes to an end. Hence, it isn't possible to access inner namespace objects from an outer namespace.



3. What is Scope Resolution in Python?

Sometimes objects within the same scope have the same name but function differently. In such cases, scope resolution comes into play in Python automatically. A few examples of such behavior are:

- Python modules namely 'math' and 'cmath' have a lot of functions that are common to both of them - log10(), acos(), exp() etc. To resolve this ambiguity, it is necessary to prefix them with their respective module, like math.exp() and cmath.exp().
- Consider the code below, an object temp has been initialized to 10 globally and then to 20 on function call. However, the function call didn't change the value of the temp globally. Here, we can observe that Python draws a clear line between global and local variables, treating their namespaces as separate identities.

Example:

```
temp = 10  # global-scope variable
```

```
def func():
    temp = 20  # local-scope variable
    print(temp)
print(temp)  # output => 10
func()  # output => 20

print(temp)  # output => 10
```

This behavior can be overridden using the global keyword inside the function, as shown in the following example:

```
temp = 10  # global-scope variable
def func():
    global temp
    temp = 20  # local-scope variable
    print(temp)
print(temp)  # output => 10
func()  # output => 20

print(temp)  # output => 20
```

4. 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
# decorator function to split words
def splitter_decorator(function):
  def wrapper():
      func = function()
      string_split = func.split()
      return string_split
  return wrapper
@splitter_decorator # this is executed next
```

```
@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!'
```

5. What are Dict and List comprehensions?

Python comprehensions, like decorators, are **syntactic sugar** constructs that help **build altered** and **filtered lists**, dictionaries, or sets from a given list, dictionary, or set. Using comprehensions saves a lot of time and code that might be considerably more verbose (containing more lines of code). Let's check out some examples, where comprehensions can be truly beneficial:

- **Performing mathematical operations on the entire list**

```
my_list = [2, 3, 5, 7, 11]
squared_list = [x**2 for x in my_list]  # list comprehension
# output => [4 , 9 , 25 , 49 , 121]
squared_dict = {x:x**2 for x in my_list}  # dict comprehension
# output => {11: 121, 2: 4 , 3: 9 , 5: 25 , 7: 49}
```

- **Performing conditional filtering operations on the entire list**

```
my_list = [2, 3, 5, 7, 11]
squared_list = [x**2 for x in my_list if x%2 != 0]  # list comprehension
# output => [9 , 25 , 49 , 121]
squared_dict = {x:x**2 for x in my_list if x%2 != 0}  # dict comprehension
# output => {11: 121, 3: 9 , 5: 25 , 7: 49}
```

- **Combining multiple lists into one**

Comprehensions allow for multiple iterators and hence, can be used to combine multiple lists into one.

```
a = [1, 2, 3]
b = [7, 8, 9]
[(x + y) for (x,y) in zip(a,b)] # parallel iterators
# output => [8, 10, 12]
[(x,y) for x in a for y in b]  # nested iterators

# output => [(1, 7), (1, 8), (1, 9), (2, 7), (2, 8), (2, 9), (3, 7), (3, 8), (3, 9)]
```

- **Flattening a multi-dimensional list**

A similar approach of nested iterators (as above) can be applied to flatten a multi-dimensional list or work upon its inner elements.

```
my_list = [[10,20,30],[40,50,60],[70,80,90]]
flattened = [x for temp in my_list for x in temp]

# output => [10, 20, 30, 40, 50, 60, 70, 80, 90]
```

Note: List comprehensions have the same effect as the map method in other languages. They follow the mathematical set builder notation rather than map and filter functions in Python.

6. 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
```

7. 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]
```

8. What is the difference between xrange and range in Python?

xrange() and **range()** are quite similar in terms of functionality. They both generate a sequence of integers, with the only difference that range() returns a **Python list**, whereas, xrange() returns an **xrange object**.

So how does that make a difference? It sure does, because unlike range(), xrange() doesn't generate a static list, it creates the value on the go. This technique is commonly used with an object-type **generator** and has been termed as "**yielding**".

Yielding is crucial in applications where memory is a constraint. Creating a static list as in range() can lead to a Memory Error in such conditions, while, xrange() can handle it optimally by using just enough memory for the generator (significantly less in comparison).

```
for i in xrange(10):    # numbers from o to 9
  print i       # output => 0 1 2 3 4 5 6 7 8 9
for i in xrange(1,10):    # numbers from 1 to 9
  print i       # output => 1 2 3 4 5 6 7 8 9
for i in xrange(1, 10, 2):    # skip by two for next
  print i       # output => 1 3 5 7 9
```

Note: **xrange** has been **deprecated** as of **Python 3.x**. Now range does exactly the same as what xrange used to do in **Python 2.x**, since it was way better to use xrange() than the original range() function in Python 2.x.

9. 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:

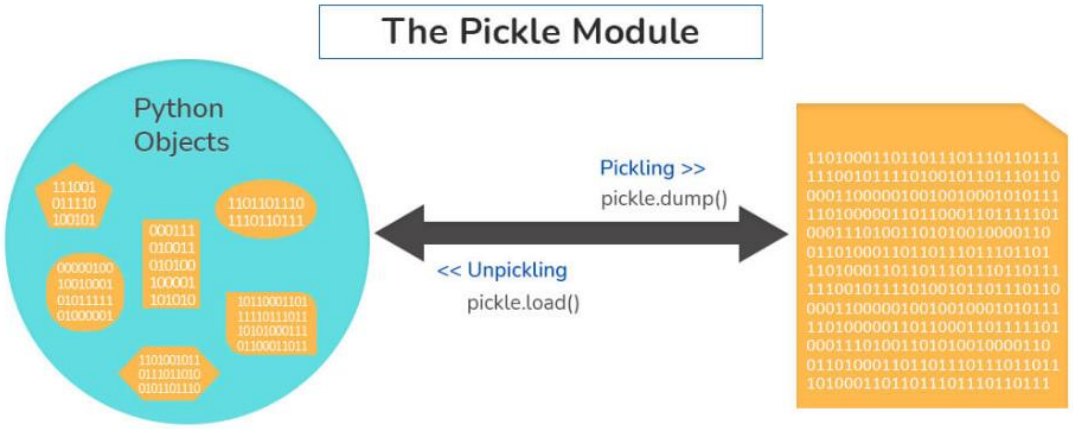
- Pickling is the name of the serialization process in Python. Any object in Python can be serialized into a byte stream and dumped as a file in the memory. The process of pickling is compact but pickle objects can be compressed further. Moreover, pickle keeps track of the objects it has serialized and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

Note: *Python has another, more primitive, serialization module called* ***marshall****, which exists primarily to* ***support .pyc files*** *in Python and* ***differs significantly from the pickle****.*



10. 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
```

11. What is PYTHONPATH in Python?

PYTHONPATH is an environment variable which you can set to add additional directories where Python will look for modules and packages. This is especially useful in maintaining Python libraries that you do not wish to install in the global default location.

12. 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.

13. 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.

14. How Python is interpreted?

- Python as a language is not interpreted or compiled. Interpreted or compiled is the property of the implementation. Python is a bytecode(set of interpreter readable instructions) interpreted generally.
- Source code is a file with .py extension.
- Python compiles the source code to a set of instructions for a virtual machine. The Python interpreter is an implementation of that virtual machine. This intermediate format is called “bytecode”.
- .py source code is first compiled to give .pyc which is bytecode. This bytecode can be then interpreted by the official CPython or JIT(Just in Time compiler) compiled by PyPy.

15. 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.

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

```
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]
```

16. What are iterators in Python?

- An iterator is an object.

- It remembers its state i.e., where it is during iteration (see code below to see how)
- __iter__() method initializes an iterator.
- It has a __next__() method which returns the next item in iteration and points to the next element. Upon reaching the end of iterable object __next__() must return StopIteration exception.
- It is also self-iterable.
- Iterators are objects with which we can iterate over iterable objects like lists, strings, etc.

```
class ArrayList:
  def __init__(self, number_list):
    self.numbers = number_list
  def __iter__(self):
    self.pos = 0
    return self
  def __next__(self):
    if(self.pos < len(self.numbers)):
      self.pos += 1
      return self.numbers[self.pos - 1]
    else:
      raise StopIteration
array_obj = ArrayList([1, 2, 3])
it = iter(array_obj)
print(next(it)) #output: 2
print(next(it)) #output: 3
print(next(it))
#Throws Exception
#Traceback (most recent call last):
#...
#StopIteration
```

17. Explain how to delete a file in Python?

Use command **os.remove(file_name)**

```
import os
os.remove("ChangedFile.csv")
print("File Removed!")
```

18. 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."
string_list = string.split(' ') #delimiter is ‘space’ character or ‘ ‘
print(string_list) #output: ['This', 'is', 'a', 'string.']
print(' '.join(string_list)) #output: This is a string.
```

19. 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.

```
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
```

**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.

```
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
```

20. What are negative indexes and why are they used?

- Negative indexes are the indexes from the end of the list or tuple or string.
- **Arr[-1]** means the last element of array **Arr[]**

```
arr = [1, 2, 3, 4, 5, 6]
#get the last element
print(arr[-1]) #output 6
#get the second last element
print(arr[-2]) #output 5
```

Python OOPS Interview Questions

1. How do you create a class in Python?

To create a class in python, we use the keyword “class” as shown in the example below:

```
class InterviewbitEmployee:
  def __init__(self, emp_name):
     self.emp_name = emp_name
```

To instantiate or create an object from the class created above, we do the following:

```
emp_1=InterviewbitEmployee("Mr. Employee")
```

To access the name attribute, we just call the attribute using the dot operator as shown below:

```
print(emp_1.emp_name)
# Prints Mr. Employee
```

To create methods inside the class, we include the methods under the scope of the class as shown below:

```
class InterviewbitEmployee:
  def __init__(self, emp_name):
     self.emp_name = emp_name

  def introduce(self):
     print("Hello I am " + self.emp_name)
```

The self parameter in the init and introduce functions represent the reference to the current class instance which is used for accessing attributes and methods of that class. The self parameter has to be the first parameter of any method defined inside the class. The method of the class InterviewbitEmployee can be accessed as shown below:

```
emp_1.introduce()
```

The overall program would look like this:

```
class InterviewbitEmployee:
```

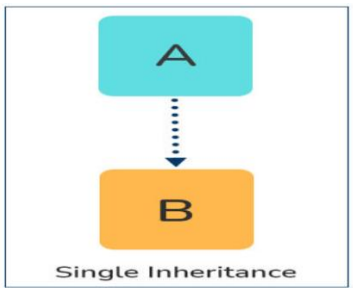
```
  def __init__(self, emp_name):
      self.emp_name = emp_name

  def introduce(self):
      print("Hello I am " + self.emp_name)

# create an object of InterviewbitEmployee class
emp_1 = InterviewbitEmployee("Mr Employee")
print(emp_1.emp_name)    #print employee name
emp_1.introduce()        #introduce the employee
```

2. How does inheritance work in python? Explain it with an example.

Inheritance gives the power to a class to access all attributes and methods of another class. It aids in code reusability and helps the developer to maintain applications without redundant code. The class inheriting from another class is a child class or also called a derived class. The class from which a child class derives the members are called parent class or superclass.

Python supports different kinds of inheritance, they are:

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

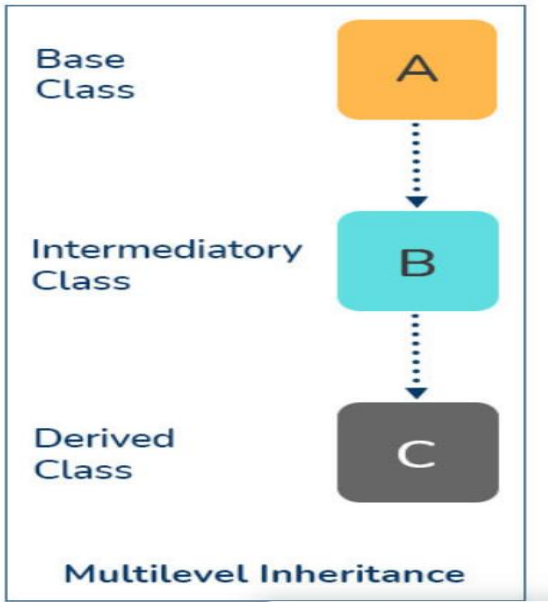


Example:

```
# 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.

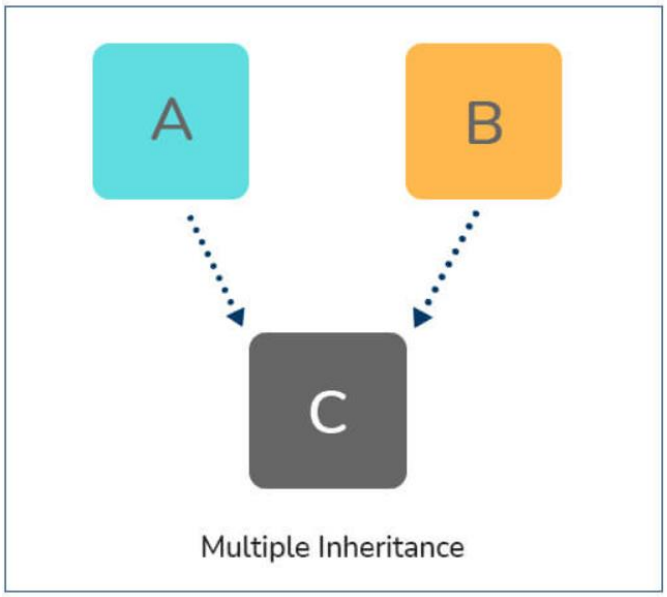


Example:

```
# 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.

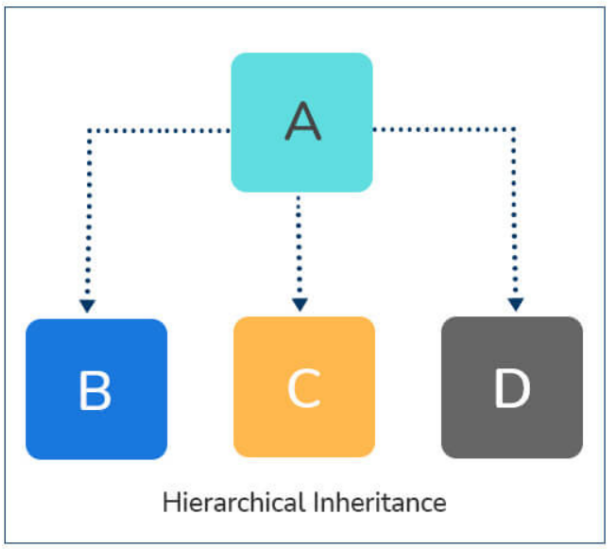


Example:

```
# 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()
```

- **Hierarchical Inheritance:** When a parent class is derived by more than one child class, it is called hierarchical inheritance.



Example:

```
# Base class
class A:
    def a_func(self):
        print("I am from the parent class.")

# 1st Derived class
class B(A):
    def b_func(self):
```

```
        print("I am from the first child.")

# 2nd Derived class
class C(A):
    def c_func(self):
        print("I am from the second child.")

# Driver's code
obj1 = B()
obj2 = C()
obj1.a_func()
obj1.b_func()   #child 1 method
obj2.a_func()

obj2.c_func()   #child 2 method
```

3. How do you access parent members in the child class?

Following are the ways using which you can access parent class members within a child class:

- **By using Parent class name:** You can use the name of the parent class to access the attributes as shown in the example below:

```
class Parent(object):
  # Constructor
  def __init__(self, name):
      self.name = name

class Child(Parent):
  # Constructor
  def __init__(self, name, age):
      Parent.name = name
      self.age = age

  def display(self):
      print(Parent.name, self.age)

# Driver Code
obj = Child("Interviewbit", 6)
obj.display()
```

- **By using super():** The parent class members can be accessed in child class using the super keyword.

```
class Parent(object):
  # Constructor
  def __init__(self, name):
     self.name = name

class Child(Parent):
  # Constructor
  def __init__(self, name, age):
     '''
      In Python 3.x, we can also use super().__init__(name)
     '''
     super(Child, self).__init__(name)
     self.age = age

  def display(self):
    # Note that Parent.name cant be used
    # here since super() is used in the constructor
    print(self.name, self.age)

# Driver Code
obj = Child("Interviewbit", 6)
obj.display()
```

4. Are access specifiers used in python?

Python does not make use of access specifiers specifically like private, public, protected, etc. However, it does not derive this from any variables. It has the concept of imitating the behaviour of variables by making use of a single (protected) or double underscore (private) as prefixed to the variable names. By default, the variables without prefixed underscores are public.

Example:

```
# to demonstrate access specifiers
class InterviewbitEmployee:

   # protected members
   _emp_name = None
   _age = None
```

```
    # private members
    __branch = None

    # constructor
    def __init__(self, emp_name, age, branch):
        self._emp_name = emp_name
        self._age = age
        self.__branch = branch

    #public member
    def display():
       print(self._emp_name +" "+self._age+" "+self.__branch)
```

5. Is it possible to call parent class without its instance creation?

Yes, it is possible if the base class is instantiated by other child classes or if the base class is a static method.

6. How is an empty class created in python?

An empty class does not have any members defined in it. It is created by using the pass keyword (the pass command does nothing in python). We can create objects for this class outside the class.
For example-

```
class EmptyClassDemo:
   pass
obj=EmptyClassDemo()
obj.name="Interviewbit"
print("Name created= ",obj.name)
```

Output:
Name created = Interviewbit

7. Differentiate between new and override modifiers.

The new modifier is used to instruct the compiler to use the new implementation and not the base class function. The Override modifier is useful for overriding a base class function inside the child class.

8. Why is finalize used?

Finalize method is used for freeing up the unmanaged resources and clean up before the garbage collection method is invoked. This helps in performing memory management tasks.

9. What is init method in python?

The **init** method works similarly to the constructors in Java. The method is run as soon as an object is instantiated. It is useful for initializing any attributes or default behaviour of the object at the time of instantiation.
For example:

```
class InterviewbitEmployee:

  # init method / constructor
  def __init__(self, emp_name):
     self.emp_name = emp_name

  # introduce method
  def introduce(self):
     print('Hello, I am ', self.emp_name)

emp = InterviewbitEmployee('Mr Employee')   # __init__ method is called here and initializes the object name with "Mr Employee"
emp.introduce()
```

10. How will you check if a class is a child of another class?

This is done by using a method called **issubclass()** provided by python. The method tells us if any class is a child of another class by returning true or false accordingly.
For example:

```
class Parent(object):
  pass

class Child(Parent):
  pass

# Driver Code
print(issubclass(Child, Parent))   #True
print(issubclass(Parent, Child))   #False
```

- We can check if an object is an instance of a class by making use of **isinstance()** method:

```
obj1 = Child()
obj2 = Parent()
print(isinstance(obj2, Child))    #False
print(isinstance(obj2, Parent))   #True
```

Python Pandas Interview Questions

1. What do you know about pandas?

- Pandas is an open-source, python-based library used in data manipulation applications requiring high performance. The name is derived from “Panel Data” having multidimensional data. This was developed in 2008 by Wes McKinney and was developed for data analysis.
- Pandas are useful in performing 5 major steps of data analysis - Load the data, clean/manipulate it, prepare it, model it, and analyze the data.

2. Define pandas dataframe.

A dataframe is a 2D mutable and tabular structure for representing data labelled with axes - rows and columns.
The syntax for creating dataframe:

```
import pandas as pd
dataframe = pd.DataFrame( data, index, columns, dtype)
```

where:

- data - Represents various forms like series, map, ndarray, lists, dict etc.
- index - Optional argument that represents an index to row labels.
- columns - Optional argument for column labels.
- Dtype - the data type of each column. Again optional.

3. How will you combine different pandas dataframes?

The dataframes can be combines using the below approaches:

- **append() method**: This is used to stack the dataframes horizontally. Syntax:

```
df1.append(df2)
```

- **concat() method:** This is used to stack dataframes vertically. This is best used when the dataframes have the same columns and similar fields. Syntax:

```
pd.concat([df1, df2])
```

- **join() method:** This is used for extracting data from various dataframes having one or more common columns.

```
df1.join(df2)
```

4. Can you create a series from the dictionary object in pandas?

One dimensional array capable of storing different data types is called a series. We can create pandas series from a dictionary object as shown below:

```
import pandas as pd
dict_info = {'key1' : 2.0, 'key2' : 3.1, 'key3' : 2.2}
series_obj = pd.Series(dict_info)
print (series_obj)
Output:
x    2.0
y    3.1
z    2.2
dtype: float64
```

If an index is not specified in the input method, then the keys of the dictionaries are sorted in ascending order for constructing the index. In case the index is passed, then values of the index label will be extracted from the dictionary.

5. How will you identify and deal with missing values in a dataframe?

We can identify if a dataframe has missing values by using the isnull() and isna() methods.

```
missing_data_count=df.isnull().sum()
```

We can handle missing values by either replacing the values in the column with 0 as follows:

```
df[‘column_name’].fillna(0)
```

Or by replacing it with the mean value of the column

```
df[‘column_name’] =
df[‘column_name’].fillna((df[‘column_name’].mean()))
```

6. What do you understand by reindexing in pandas?

Reindexing is the process of conforming a dataframe to a new index with optional filling logic. If the values are missing in the previous index, then NaN/NA is placed in the location. A new object is returned unless a new index is produced that is equivalent to the current one. The copy value is set to False. This is also used for changing the index of rows and columns in the dataframe.

7. How to add new column to pandas dataframe?

A new column can be added to a pandas dataframe as follows:

```
import pandas as pd
data_info = {'first' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
      'second' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}

df = pd.DataFrame(data_info)
#To add new column third
df['third']=pd.Series([10,20,30],index=['a','b','c'])
print (df)
#To add new column fourth
df['fourth']=df['first']+info['third']
print (df)
```

8. How will you delete indices, rows and columns from a dataframe?

To delete an Index:

- Execute del df.index.name for removing the index by name.
- Alternatively, the df.index.name can be assigned to None.
- For example, if you have the below dataframe:

```
        Column 1
Names
John         1
Jack         2
Judy         3
Jim          4
```

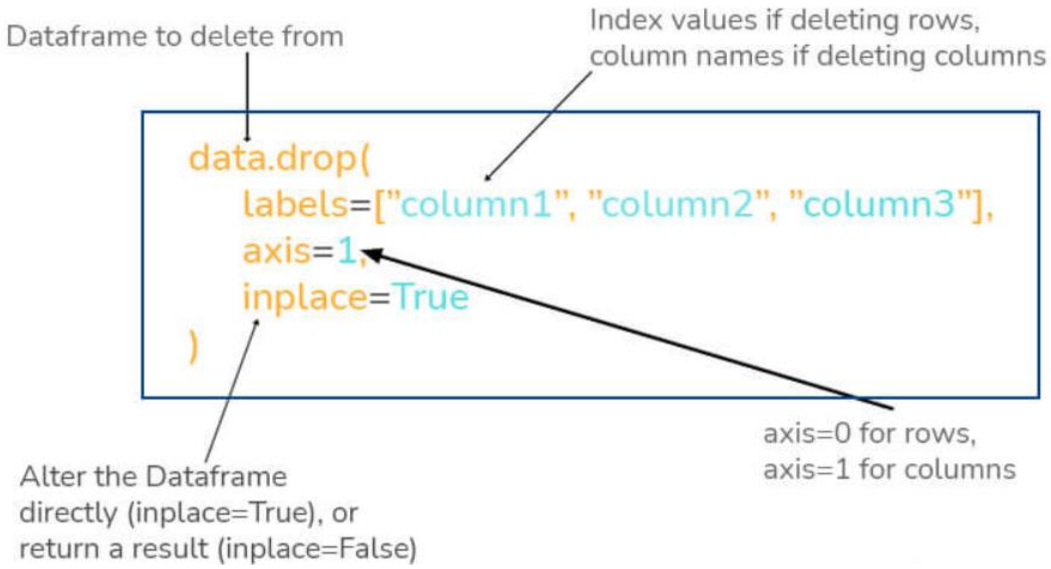
- To drop the index name “Names”:

```
df.index.name = None
# Or run the below:
# del df.index.name
```

```
print(df)
      Column 1
John         1
Jack         2
Judy         3
Jim          4
```

To delete row/column from dataframe:

- drop() method is used to delete row/column from dataframe.
- The axis argument is passed to the drop method where if the value is 0, it indicates to drop/delete a row and if 1 it has to drop the column.
- Additionally, we can try to delete the rows/columns in place by setting the value of inplace to True. This makes sure that the job is done without the need for reassignment.
- The duplicate values from the row/column can be deleted by using the drop_duplicates() method.



9. Can you get items of series A that are not available in another series B?

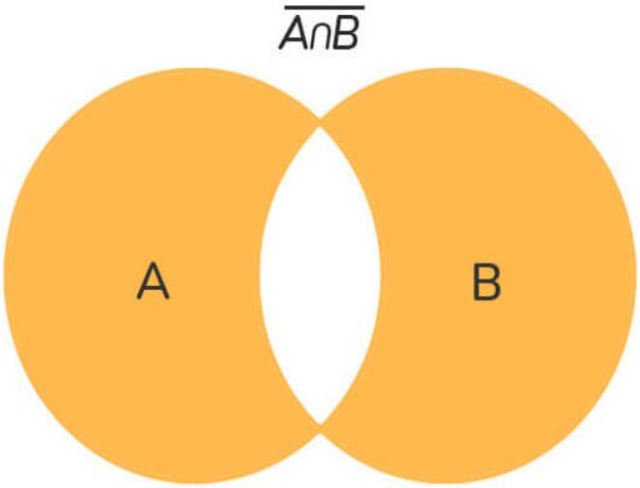
This can be achieved by using the ~ (not/negation symbol) and isin() method as shown below.

```
import pandas as pd
df1 = pd.Series([2, 4, 8, 10, 12])
df2 = pd.Series([8, 12, 10, 15, 16])
```

```
df1=df1[~df1.isin(df2)]
print(df1)
"""
Output:
0    2
1    4
dtype: int64
"""
```

10. How will you get the items that are not common to both the given series A and B?

We can achieve this by first performing the union of both series, then taking the intersection of both series. Then we follow the approach of getting items of union that are not there in the list of the intersection.



The following code demonstrates this:

```
import pandas as pd
import numpy as np
df1 = pd.Series([2, 4, 5, 8, 10])
df2 = pd.Series([8, 10, 13, 15, 17])
p_union = pd.Series(np.union1d(df1, df2))  # union of series
p_intersect = pd.Series(np.intersect1d(df1, df2))  # intersection of series
unique_elements = p_union[~p_union.isin(p_intersect)]
print(unique_elements)
"""
Output:
0     2
```

```
1    4
2    5
5   13
6   15
7   17
dtype: int64
"""
```

11. While importing data from different sources, can the pandas library recognize dates?

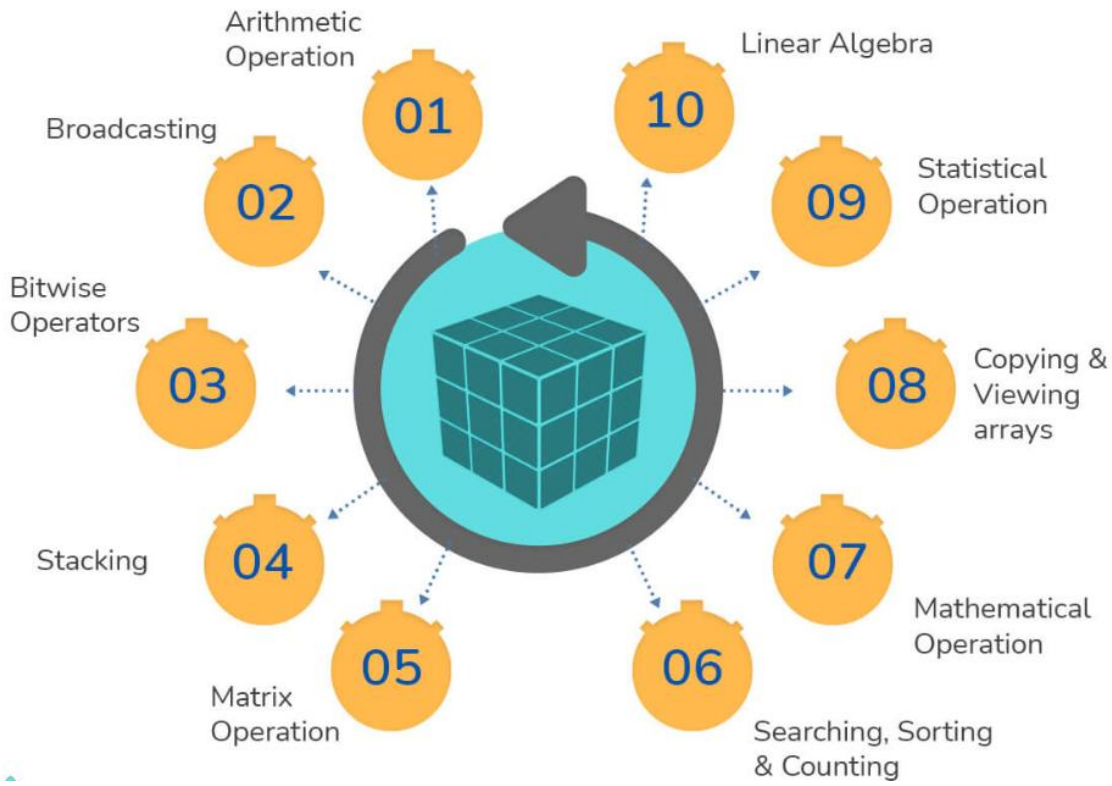
Yes, they can, but with some bit of help. We need to add the parse_dates argument while we are reading data from the sources. Consider an example where we read data from a CSV file, we may encounter different date-time formats that are not readable by the pandas library. In this case, pandas provide flexibility to build our custom date parser with the help of lambda functions as shown below:

```
import pandas as pd
from datetime import datetime
dateparser = lambda date_val: datetime.strptime(date_val, '%Y-%m-%d %H:%M:%S')
df = pd.read_csv("some_file.csv", parse_dates=['datetime_column'], date_parser=dateparser)
```

Numpy Interview Questions

1. What do you understand by NumPy?

NumPy is one of the most popular, easy-to-use, versatile, open-source, python-based, general-purpose package that is used for processing arrays. NumPy is short for NUMerical PYthon. This is very famous for its highly optimized tools that result in high performance and powerful N-Dimensional array processing feature that is designed explicitly to work on complex arrays. Due to its popularity and powerful performance and its flexibility to perform various operations like trigonometric operations, algebraic and statistical computations, it is most commonly used in performing scientific computations and various broadcasting functions. The following image shows the applications of NumPy:



2. How are NumPy arrays advantageous over python lists?

- The list data structure of python is very highly efficient and is capable of performing various functions. But, they have severe limitations when it comes to the computation of vectorized operations which deals with element-wise multiplication and addition. The python lists also require the information regarding the type of every element which results in overhead as type dispatching code gets executes

every time any operation is performed on any element. This is where the NumPy arrays come into the picture as all the limitations of python lists are handled in NumPy arrays.

- Additionally, as the size of the NumPy arrays increases, NumPy becomes around 30x times faster than the Python List. This is because the Numpy arrays are densely packed in the memory due to their homogenous nature. This ensures the memory free up is also faster.

3. What are the steps to create 1D, 2D and 3D arrays?

- **1D array creation:**

```
import numpy as np
one_dimensional_list = [1,2,4]
one_dimensional_arr = np.array(one_dimensional_list)
print("1D array is : ",one_dimensional_arr)
```

- **2D array creation:**

```
import numpy as np
two_dimensional_list=[[1,2,3],[4,5,6]]
two_dimensional_arr = np.array(two_dimensional_list)
print("2D array is : ",two_dimensional_arr)
```

- **3D array creation:**

```
import numpy as np
three_dimensional_list=[[[1,2,3],[4,5,6],[7,8,9]]]
three_dimensional_arr = np.array(three_dimensional_list)
print("3D array is : ",three_dimensional_arr)
```

- **ND array creation:** This can be achieved by giving the ndmin attribute. The below example demonstrates the creation of a 6D array:

```
import numpy as np
ndArray = np.array([1, 2, 3, 4], ndmin=6)
print(ndArray)
print('Dimensions of array:', ndArray.ndim)
```

4. You are given a numpy array and a new column as inputs. How will you delete the second column and replace the column with a new column value?

Example:
Given array:

```
[[35 53 63]
[72 12 22]
[43 84 56]]
```

New Column values:

```
[
  20
  30
  40
]
```

Solution:

```
import numpy as np
#inputs
inputArray = np.array([[35,53,63],[72,12,22],[43,84,56]])
new_col = np.array([[20,30,40]])
# delete 2nd column
arr = np.delete(inputArray , 1, axis = 1)
#insert new_col to array
arr = np.insert(arr , 1, new_col, axis = 1)
print (arr)
```

5. How will you efficiently load data from a text file?

We can use the method numpy.loadtxt() which can automatically read the file's header and footer lines and the comments if any.

This method is highly efficient and even if this method feels less efficient, then the data should be represented in a more efficient format such as CSV etc. Various alternatives can be considered depending on the version of NumPy used.

Following are the file formats that are supported:

- Text files: These files are generally very slow, huge but portable and are human-readable.
- Raw binary: This file does not have any metadata and is not portable. But they are fast.

- Pickle: These are borderline slow and portable but depends on the NumPy versions.
- HDF5: This is known as the High-Powered Kitchen Sink format which supports both PyTables and h5py format.
- .npy: This is NumPy's native binary data format which is extremely simple, efficient and portable.

6. How will you read CSV data into an array in NumPy?

This can be achieved by using the genfromtxt() method by setting the delimiter as a comma.

```
from numpy import genfromtxt
csv_data = genfromtxt('sample_file.csv', delimiter=',')
```

7. How will you sort the array based on the Nth column?

For example, consider an array arr.

```
arr = np.array([[8, 3, 2],
        [3, 6, 5],
        [6, 1, 4]])
```

Let us try to sort the rows by the 2nd column so that we get:

```
[[6, 1, 4],
[8, 3, 2],
[3, 6, 5]]
```

We can do this by using the sort() method in numpy as:

```
import numpy as np
arr = np.array([[8, 3, 2],
        [3, 6, 5],
        [6, 1, 4]])
#sort the array using np.sort
arr = np.sort(arr.view('i8,i8,i8'),
      order=['f1'],
      axis=0).view(np.int)
```

We can also perform sorting and that too inplace sorting by doing:

```
arr.view('i8,i8,i8').sort(order=['f1'], axis=0)
```

8. How will you find the nearest value in a given numpy array?

We can use the argmin() method of numpy as shown below:

```
import numpy as np
def find_nearest_value(arr, value):
   arr = np.asarray(arr)
   idx = (np.abs(arr - value)).argmin()
   return arr[idx]
#Driver code
arr = np.array([ 0.21169,  0.61391, 0.6341, 0.0131, 0.16541,  0.5645,
0.5742])
value = 0.52
print(find_nearest_value(arr, value)) # Prints 0.5645
```

9. How will you reverse the numpy array using one line of code?

This can be done as shown in the following:

```
reversed_array = arr[::-1]
```

where **arr** = original given array, reverse_array is the resultant after reversing all elements in the input.

10. How will you find the shape of any given NumPy array?

We can use the shape attribute of the numpy array to find the shape. It returns the shape of the array in terms of row count and column count of the array.

```
import numpy as np
arr_two_dim = np.array([("x1","x2", "x3","x4"),
            ("x5","x6", "x7","x8" )])
arr_one_dim = np.array([3,2,4,5,6])
# find and print shape
print("2-D Array Shape: ", arr_two_dim.shape)
print("1-D Array Shape: ", arr_one_dim.shape)
"""
Output:
2-D Array Shape:  (2, 4)
1-D Array Shape:  (5,)
"""
```

Python Libraries Interview Questions

1. Differentiate between a package and a module in python.

The module is a single python file. A module can import other modules (other python files) as objects. Whereas, a package is the folder/directory where different sub-packages and the modules reside.

A python module is created by saving a file with the extension of .py. This file will have classes and functions that are reusable in the code as well as across modules.

A python package is created by following the below steps:

- Create a directory and give a valid name that represents its operation.
- Place modules of one kind in this directory.
- Create __init__.py file in this directory. This lets python know the directory we created is a package. The contents of this package can be imported across different modules in other packages to reuse the functionality.

2. What are some of the most commonly used built-in modules in Python?

Python modules are the files having python code which can be functions, variables or classes. These go by .py extension. The most commonly available built-in modules are:

- os
- math
- sys
- random
- re
- datetime
- JSON

3. What are lambda functions?

Lambda functions are generally inline, anonymous functions represented by a single expression. They are used for creating function objects during runtime. They can accept any number of parameters. They are usually used where functions are required only for a short period. They can be used as:

```
mul_func = lambda x,y : x*y
print(mul_func(6, 4))
# Output: 24
```

4. How can you generate random numbers?

Python provides a module called random using which we can generate random numbers.

- We have to import a random module and call the random() method as shown below:
  - The random() method generates float values lying between 0 and 1 randomly.

```
import random
print(random.random())
```

- To generate customised random numbers between specified ranges, we can use the randrange() method
  Syntax: randrange(beginning, end, step)
  For example:

```
import random
print(random.randrange(5,100,2))
```

5. Can you easily check if all characters in the given string is alphanumeric?

This can be easily done by making use of the isalnum() method that returns true in case the string has only alphanumeric characters.

For Example -

```
"abdc1321".isalnum() #Output: True
"xyz@123$".isalnum() #Output: False
```

Another way is to use match() method from the re (regex) module as shown:

```
import re
print(bool(re.match('[A-Za-z0-9]+$','abdc1321'))) # Output: True
print(bool(re.match('[A-Za-z0-9]+$','xyz@123$'))) # Output: False
```

6. What are the differences between pickling and unpickling?

Pickling is the conversion of python objects to binary form. Whereas, unpickling is the conversion of binary form data to python objects. The pickled objects are used for storing in disks or external memory locations. Unpickled objects are used for getting the data back as python objects upon which processing can be done in python.

Python provides a pickle module for achieving this. Pickling uses the pickle.dump() method to dump python objects into disks. Unpickling uses the pickle.load() method to get back the data as python objects.

7. Define PYTHONPATH.

It is an environment variable used for incorporating additional directories during the import of a module or a package. PYTHONPATH is used for checking if the imported packages or modules are available in the existing directories. Not just that, the interpreter uses this environment variable to identify which module needs to be loaded.

8. Define PIP.

PIP stands for Python Installer Package. As the name indicates, it is used for installing different python modules. It is a command-line tool providing a seamless interface for installing different python modules. It searches over the internet for the package and installs them into the working directory without the need for any interaction with the user. The syntax for this is:

```
pip install <package_name>
```

9. Are there any tools for identifying bugs and performing static analysis in python?

Yes, there are tools like PyChecker and Pylint which are used as static analysis and linting tools respectively. PyChecker helps find bugs in python source code files and raises alerts for code issues and their complexity. Pylint checks for the module's coding standards and supports different plugins to enable custom features to meet this requirement.

10. Differentiate between deep and shallow copies.

- Shallow copy does the task of creating new objects storing references of original elements. This does not undergo recursion to create copies of nested objects. It just copies the reference details of nested objects.

- Deep copy creates an independent and new copy of an object and even copies all the nested objects of the original element recursively.

11. What is main function in python? How do you invoke it?

In the world of programming languages, the main is considered as an entry point of execution for a program. But in python, it is known that the interpreter serially interprets the file line-by-line. This means that python does not provide main() function explicitly. But this doesn't mean that we cannot simulate the execution of main. This can be done by defining user-defined main() function and by using the __name__ property of python file. This __name__ variable is a special built-in variable that points to the name of the current module. This can be done as shown below:

```
def main():
   print("Hi Interviewbit!")
if __name__=="__main__":
   main()
```